个性化你的阅读

编程狂人

Programming Madman

NO.8

推酷

关于推酷

推酷是专注于 IT 圈的个性化阅读社区。我们利用智能算法，从海量文章资讯中挖掘出高质量的内容，并通过分析用户的阅读偏好，准实时推荐给你最感兴趣的内容。我们推荐的内容包含科技、创业、设计、技术、营销等多方面内容，满足你日常的专业阅读需要。我们针对 IT 人还做了个活动频道，它聚合了 IT 圈最新最全的线上线下活动，使 IT 人能更方便地找到感兴趣的活动信息。

关于周刊

推酷周刊是专为 IT 人打造的行业技术周刊，目前推出的《编程狂人》是献给 广大的程序员们。 我们利用技术挖掘出那些高质量的文章，并通过人工加以筛 选整理出来。 每期的周刊一般会周一的某个时间点发布。

联系我们

tuicool2012　　164644910　　推酷网

下载 APP

Android版本

立即下载 Android

iPhone版本

免费下载就在 App Store

2014/01/13/第八期

目录

后端架构

程序人生

正文

[业界新闻]

特斯拉升级充电软件和适配器：预防车辆着火

美国电动汽车制造商特斯拉今天宣布，该公司已经采取了一些措施来防止充电系统过热，包括提供新型墙式适配器，并对充电软件进行升级。

去年11月，美国加州欧文市曾经发生过一起与特斯拉 Model S 汽车有关的车库着火事件。当地消防部门表示，那场火灾可能是由特斯拉的充电系统或车库墙上的插座面板引发的。

特斯拉当时不认同这一结论，否认该公司的充电线路与火灾有关。特斯拉发言人尚未对此次升级是否与那起火灾有关发表评论。

特斯拉在新闻稿中表示，该公司的目标是防止汽车的充电适配器过热，并指出，电路板年久腐蚀或接线不当等原因都会导致过热。

通过去年12月对充电软件进行的调整，一旦充电系统探测到电流波动，便会将充电强度降低25%。

该公司在新闻稿中说：“特斯拉相信，此次软件更新将完全解决所有风险。”但作为预防措施，该公司还是会在几周内为受影响的用户提供内置保险丝新版墙式适配器。

另外，近几个月先后发生了三起 Model S 汽车道路着火事故，导致特斯拉股价在去年10月暴跌。

原文链接：http://news.mydrivers.com/1/289/289573.htm?utm_source=tuicool

Tengine-2.0.0 正式发布，淘宝的 Nginx 增强版

这个版本中包括了对 SPDY v3协议的实现，其中最重要的是支持了 SPDY 流控。我们还增强 DSO 模块，编译动态模块不再依赖原始编译环境了。另外，这个版本基于的 Nginx 核心也迁移到了1.4.4。

完整的更新列表如下：

* Feature: 增强 DSO 模块，编译动态模块不再依赖原始编译环境 [monadbobo]

* Feature: 支持 SPDY v3协议，自动检测同一端口的 SPDY 请求和 HTTP 请求 [lilbedwin、chobits]

* Feature: 支持设置 proxy、memcached、fastcgi、scgi、uwsgi 在后端失败时的重试次数 [supertcy]

* Feature: tfs 模块在 RcServer 心跳时汇报访问统计 [zhcn381]

* Feature: if 指令支持比较数值大小：'>'、'<'、'>='、'<=' [flygoast]

* Feature: 健康检查模块支持长连接检查 [lilbedwin]

* Feature: trim 模块支持 SSI 和 ESI 的注释 [taoyuanyuan]

* Feature: expires_by_types 指令支持使用通配符，例如'text/*'匹配子类型 [zhcn381]

* Feature: 增加$base64_decode_变量前缀，支持计算指定变量的 base64解压结果 [yzprofile]

* Feature: 增加$md5_encode_变量前缀，支持计算指定变量的 md5哈希 [yzprofile]

* Feature: 增加$time_http 变量，支持按 http 格式输出当前时间 [flygoast]

* Feature: 增加$full_request 变量，取得原始的请求 url，包括协议类型和域名 [yzprofile]

* Feature: 增加$escape_uri_变量前缀，支持对指定变量进行 url 转义 [yzprofile]

* Feature: 增加$raw_uri 变量，支持取得不含参数的原始 uri [flygoast]

* Feature: 支持按微秒记录子请求的请求时间 [jinglong]

* Feature: 增加 API，支持对 url 进行 base64编码 [lilbedwin]
* Change: 合并 nginx-1.4.4版本的修改 [cfsego]
* Change: 修改 stub_status 模块，不对子请求进行统计 [jinglong]
* Bugfix: 修正 footer 模块，不处理含有 Content-Encoding 头的响应 [yaoweibin]
* Bugfix: 修正 client_body_postpone_size 指令设置为0时出现的问题 [yaoweibin]
* Bugfix: 修正 Lua 模块编译时出现警告 [diwayou]

原文链接：http://www.oschina.net/news/47658/tengine-2-0?utm_source=tuicool

MongoDB 2.4.9 发布，NoSQL 数据库

NoSQL 数据库 MongoDB 2.4.9发布. 2014-01-11 之前版本是2013-11-01的 2.4.8 Bugfix 版本.其他产品线2.2.6 2.0.9 开发版2.5.4

MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。他支持的数据结构非常松散，是类似 json 的 bjson 格式，因此可以存储比较复杂的数据类型。 Mongo 最大的特点是他支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

它的特点是高性能、易部署、易使用，存储数据非常方便。主要功能特性有：

面向集合存储，易存储对象类型的数据。

模式自由。

支持动态查询。

支持完全索引，包含内部对象。

支持查询。

支持复制和故障恢复。

使用高效的二进制数据存储，包括大型对象（如视频等）。

自动处理碎片，以支持云计算层次的扩展性

支持 RUBY，PYTHON，JAVA，C++，PHP 等多种语言。

文件存储格式为 BSON（一种 JSON 的扩展）

可通过网络访问

完全改进：

Bug

[SERVER-5625] - New sharded connections to a namespace trigger setShardVersion on all shards

[SERVER-7246] - Mongos cannot do slaveOk queries when primary is down

[SERVER-9238] - Shell stops working after long autocomplete operation

[SERVER-10538] - Passing $where predicate to db.currentOp() crashes mongod

[SERVER-10919] - logging in ~ReplicaSetMonitor() crashes

[SERVER-11099] - clang compiled mongo shell crashes on exit with a stack trace in v8

[SERVER-11194] - Non-numeric expiresAfterSeconds causes bad TTL query

[SERVER-11494] - textIndexVersion compatibility check not complete

[SERVER-11502] - misplaced openssl callback registration can cause crashes

[SERVER-11731] - $where inside of projection $elemMatch causes segmentation fault

[SERVER-11971] - slaveok versioning logic in mongos should also apply to read prefs

[SERVER-12041] - retry logic for read preferences should also apply on lazy recv() network failure

[SERVER-12092] - Modifying collection options can cause restores of

collection to fail

[SERVER-12094] - Cannot set false setParameter options in config file

[SERVER-12146] - writeback listener may not get correct code back from ClientInfo::getLastError

Task

[SERVER-11908] - Failure to rollback usePowerOf2Sizes should not cause fatal error

Sub-task

[SERVER-11977] - Support for non-client opTime in mongod GLE

原文链接：http://www.oschina.net/news/47736/mongodb-2-4-9?utm_source=tuicool

Apache Tomcat 7.0.50 发布

Apache Tomcat 7.0.50 发布，此版本相对于上一个版本 7.0.47包括了一些 bug 修复和功能改进。

注意事项：

此版本包括4个压缩的二进制文件：一个通用文件和三个可运行在不同 CPU 结构的 windows 二进制文件；

需要 Java 7 才能使用 JSR-356 Java WebSocket 1.0 实现

如果使用 ARP/native AJP 或 HTTP 连接器，你*必须*升级到 ARP/native AJP 1.1.29 或更高版本的库。

Tomcat 服务器是一个免费的开放源代码的 Web 应用服务器。

Tomcat 是 Apache 软件基金会（Apache Software Foundation）的 Jakarta 项目中的一个核心项目，由 Apache、Sun 和其他一些公司及个人共同开发而成。由于有了 Sun 的参与和支持，最新的 Servlet 和 JSP 规范总是能在 Tomcat 中得到体现，Tomcat 5 支持最新的 Servlet 2.4 和 JSP 2.0 规范。因为 Tomcat 技

术先进、性能稳定，而且免费，因而深受 Java 爱好者的喜爱并得到了部分软件开发商的认可，成为目前比较流行的 Web 应用服务器。

Tomcat 很受广大程序员的喜欢，因为它运行时占用的系统资源小，扩展性好，支持负载平衡与邮件服务等开发应用系统常用的功能；而且它还在不断的改进和完善中，任何一个感兴趣的程序员都可以更改它或在其中加入新的功能。

原文链接：

http://www.linuxeden.com/html/versionupdate/2014/01/147434.html?utm_source=tuicool

Zz 笔者带你剖析 Java7.x 新特性

前言

最近在 ITeye 上看见一些朋友正在**激烈**讨论关于 Java7.x 的一些语法结构，所以笔者有些手痒，特此**探寻**了7.x（此篇博文笔者使用的是目前最新版本的 JDK-7u15）的一些**新特性**分享给大家。虽然目前很多开发人员至今还在沿用 Java4.x（笔者项目至今沿用4.x），但这并不是成为不前进的**借口**。想了解 Java 的发展，想探寻 Java 的未来，那么你务必需要时刻保持一颗永不落后的心。

当然笔者此篇博文**并不代表官方观点**，如果有朋友觉得笔者的话语是缪论，希望指正提出，笔者会在第一时间纠正博文内容。在此笔者先谢过各位利用宝贵的时间阅读此篇博文，最后笔者祝愿各位新年大吉，工作顺利。再次啰嗦一点，**SSJ 的系列博文，笔者将会在本周更新**，希望大家体谅。

目录

五、声明二进制字面值；

六、catch 表达式调整；

七、文件系统改变；

八、探讨 Java I/O 模型；

九、Swing Framework（JSR 296规范）支持；

十、JVM 内核升级之 Class Loader 架构调整；

十一、JVM 内核升级之 Garbage Collector 调整（时间仓促，后期讲解）；

一、自动资源管理

早在7.x 版本之前，某些可回收资源比如：I/O 链接、DB 连接、TCP/UDP 连接。开发人员都需要在使用后对其进行**手动**关闭，如果不关闭或者忘记关闭这些资源，就会长期**霸占** JVM 内部的资源，极大程度上**影响**了 JVM 的资源分配。就像内存管理一样，开发人员梦寐以求的就是希望有一天再也无需关注**繁琐**的资源管理（资源创建、资源就绪、资源回收）。值得庆幸的是7.x 为我们带来了一次彻头彻尾改变，我们将再也不必以手动管理我们的资源。

早在 Java5.x 的时候，Java API 为开发人员提供了一个 Closeable 接口。该接口中包含一个 close()方法，允许所有可回收资源的类型对其进行重写实现。7.x 版本中几乎所有的资源类型都实现了 Closeable 接口，并重写了 close()方法。也就是说所有可回收的系统资源，我们将**再不必**每次使用完后调用 close()方法进行资源回收，这一切全部交接给自动资源管理器去做即可。

例如 Reader 资源类型继承 Closeable 接口实现资源自动管理：

Java 代码

```
public abstract class Reader implements Readable, Closeable
```

当然如果你需要在程序中使用自动资源管理，还需要使用 API 提供的新语法支持，这类语法包含在 try 语句块内部。看到这里你可能不禁感叹，try 也能**支持表达式**了，是的7.x 确实允许 try 使用表达式的**语法方式**实现自动资源管理，但**仅限**于资源类型。

使用 try 表达式实现自动资源管理：

Java 代码

```
try(BufferedReader reader = new BufferedReader(new FileReader("
```

```
路径"));)
    • {
    •     //...
    • }
    • catch(Exception e)
    • {
    •     e.printStackTrace();
    • }
```

二、“<>”类型推断运算符

Java5.x 新增了许多新的功能，在这些新引入的功能中，泛型最为**重要**。泛型是一种新的语法元素，泛型的出现导致整个 Java API 都发生了变化（比如：Java 集合框架就使用了泛型语法）。

在泛型没有出现之前，我们都是将 Object 类作为通用的**任意数据类型**使用。因为在 Java 语言中，Object 类是所有类的超类。但是使用 Object 类作为任意数据类型并不是安全的，因为在很多时候我们需要将 Object 类型向下转换，在这些转换过程中偶尔也可能出现**不匹配**的类型转换错误。泛型的出现则很好的解决了 Object 类型所存在的**安全性**问题，且泛型还扩展了代码的重用性。

泛型的核心概念就是**参数化类型**，所谓参数化类型指的就是开发人员可以在**外部**指定的数据类型来创建泛型类、泛型接口和泛型方法。

使用泛型类型示例：

Java 代码

```
• List<String> list = new ArrayList<String>();
```

通过上述程序示例我们可以看出，笔者定义了一个泛型类型为 String 的 List 集合。这样一来 List 集合的泛型参数将会被定义为 String 类型。但是你有没有想过，使用里氏替换原则或者实例化泛型类型时，其实现可以简化泛型类型声明吗？答案是肯定的，在 Java7.x 中，允许使用运算符“<>”来做类型推断。也就是说你只需要在声明时标注泛型类型，实现时**无需重复**标注。

使用“<>”类型推断运算符简化泛型语法：

Java 代码

```
List<String> list = new ArrayList<>();
```

三、字面值下划线支持

不知道大家有没有过同笔者一样的烦恼，早在 Java7.x 版本之前，咱们在定义 int 或者 long 类型等变量的字面值时，往往会因为其定义的**值过长**，从而严重影响后续的**可读性**。如果你也是这么觉得，那么你可以考虑使用 Java7.x 为字面值操作提供的可读性优化。那便是允许你直接的字面值中使用符号“_”进行**切分**，这样一来不仅可以提升可读性，还能够清晰的分辨出字面值的长度。当然程序运行时自然会将“_”符号进行提取再做运算。

使用“_”符号进行字面值可读性优化：

Java 代码

```
int money = 100_000_000;
```

四、switch 字面值支持

Java 一共为开发人员提供了2种多路分支语句，一种是大家常用的 if-else，另一种则是 switch 语句。早在 Java7.x 版本之前，switch 语句表达式值只能定义 byte、short、int 和 char 等**4**种类型，且该语句表达式值**只能匹配一个**，故不能重复。但是 Java7.x 的到来允许 switch 定义另一种**全新**的表达式值，那就是 String 类型。

使用 String 类型作为 Switch 表达式值：

Java 代码

```
switch("a")
{
    case "a":
        System.out.println("a");
        break;
    case "b":
        System.out.println("b");
}
```

五、声明二进制字面值

Java 与 C 语言、C++语言直接相关。Java 语言继承了 C 语言的语法结构，而 OMT（Object Modeling Technique，对象模型）则是直接从 C++语言改编而来的。所以早在 Java7.x 版本之前，开发人员只能够定义十进制、八进制、十六进制等字面值。但是现在你完全可以使用“0b”字符为**前缀**定义二进制字面值。

定义二进制字面值：

Java 代码

```
int test = 0b010101;
```

当然这里笔者需要提示你的是，虽然咱们可以直接在程序中定义二进制字面值。但是在程序运算时，仍然会将其**转换**成十进制展开运算和输出。

六、catch 表达式调整

谈到 catch 语句的时候，不得不提到 try 语句，因为它们彼此之间存在**相互依赖、相互关联**的关系。在 Java 程序中捕获一个异常采用的是 try 和 catch 语句，try 语句里面所包含的代码块都是需要进行**异常监测**的，而 catch 语句里面所包含的代码块，则是告诉程序当异常发生的时候所需要执行的**异常处理**。

谈到捕获异常，在 Java7.x 之前有2种方式。第一种是采用定义多个 catch 代码块，另外一种则是直接使用 Exception（可恢复性异常超类）进行捕获。但是现在，如果你觉得不想**笼统**的将所有异常定义为 Exception 进行捕获，或者**纠结**于反复定义 catch 代码块，那么你完全可以采用 Java7.x 的 catch 表达式调整。Java7.x 允许你在 catch 表达式内部使用“|”运算符匹配多个异常类型，当触发异常时，异常类型将自动进行**类型匹配**操作。

使用“|”运算符定义 catch 表达式：

Java 代码

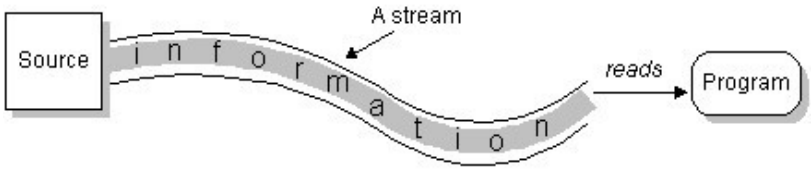
```
try
{
    //...
}
catch (SQLException | Exception e)
{
```

```
    e.printStackTrace();
}
```

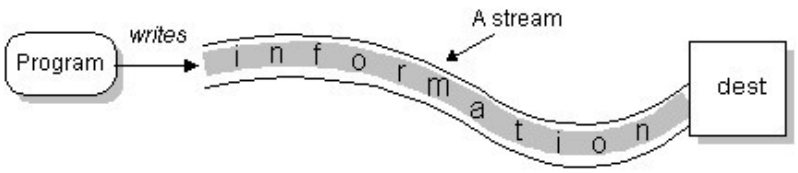
七、文件系统改变

既然本章节咱们已经谈到了 Java 的文件系统（FileSystem），那么必然同样也会关联到 I/O 技术。其实所谓 I/O（Input/Output）指的就是数据输入/输出的**过程**，我们称之为流（**数据通信通道**）这个概念。比如当 Java 应用程序需要读取目标数据源的数据时，则开启输入流。需要写入时，则开启输出流。数据源允许是本地磁盘、内存或者是网络中的数据。

向目标数据源读取数据：



向目标数据源写入数据：



Java 的文件系统主要由 java.io 及 java.nio 两个包内的组件**构成**。早在 Java7.x 之前，文件的操作一向都比较**棘手**。当然笔者这里提出的棘手，更多的是指向 Java API 对文件的管理的**不便**。比如咱们需要编写一个程序，这个程序的功能仅仅只是拷贝文件后进行粘贴。但就是连这样简单的程序逻辑实现，开发人员动则都需要编写几十行**有效代码**。

使用 Java File API 操作文件核心示例：

Java 代码

```
/* 复制目标数据源数据 */
BufferedInputStream reader = new BufferedInputStream(
        new FileInputStream(COPYFILEPATH));
```

```
byte[] content = new byte[reader.available()];
reader.read(content);

/* 将复制数据粘贴至新目录 */
BufferedOutputStream write = new BufferedOutputStream(
        new FileOutputStream(PASTEFILEPATH));
write.write(content);
```

通过上述程序示例我们可以看出，仅仅只是编写一个简单的文件复制粘贴逻辑，我们的代码量都大得惊人。如果你也认同上述程序的**繁琐**，那么你完全有必要体验下 Java7.x 对文件系统的一次全新**改变**。

Java7.x 推出了全新的 NIO.2 API 以此改变针对文件管理的不便，使得在 java.nio.file 包下使用 Path、Paths、Files、WatchService、FileSystem 等常用类型可以很好的**简化**开发人员对文件管理的编码工作。

咱们就先从 Path 接口开始进行讲解吧。Path 接口的**某些功能**其实可以和 java.io 包下的 File 类型**等价**，当然这些功能仅限于**只读操作**。在实际开发过程中，开发人员可以联用 Path 接口和 Paths 类型，从而获取文件的一系列上下文信息。

Path 接口常用方法如下：

方法名称	方法返回类型	方法描述
getNameCount()	int	获取当前文件节点数
getFileName()	java.nio.file.Path	获取当前文件名称
getRoot()	java.nio.file.Path	获取当前文件根目录
getParent()	java.nio.file.Path	获取当前文件上级关联目录

联用 Path 接口和 Paths 类型获取文件信息：

Java 代码

```
@Test
```

```
public void testFile() {
    Path path = Paths.get("路径：/文件");
    System.out.println("文件节点数:" + path.getNameCount());
    System.out.println("文件名称:" + path.getFileName());
    System.out.println("文件根目录:" + path.getRoot());
    System.out.println("文件上级关联目录:" + path.getParent());
}
```

通过上述程序示例我们可以看出，联用 Path 接口和 Paths 类型可以很方便的访问到目标文件的上下文信息。当然这些操作全都是只读的，如果开发人员想对文件进行其它**非只读**操作，比如文件的创建、修改、删除等操作，则可以使用 Files 类型进行操作。

Files 类型常用方法如下：

方法名称	方法返回类型	方法描述
createFile()	java.nio.file.Path	在指定的目标目录创建新文件
delete()	void	删除指定目标路径的文件或文件夹
copy()	java.nio.file.Path	将指定目标路径的文件拷贝到另一个文件中
move()	java.nio.file.Path	将指定目标路径的文件转移到其他路径下，并删除源文件

使用 Files 类型复制、粘贴文件示例：

Java 代码

```
Files.copy(Paths.get("路径：/源文件"), Paths.get("路径：/新文件"));
```

通过上述程序示例我们可以看出，使用 Files 类型来管理文件，相对于传统的 I/O 方式来说更加**方便**和**简单**。因为具体的操作实现将全部移交给 NIO.2 API，开发人员则无需关注。

Java7.x 还为开发人员提供了一套**全新**的文件系统功能，那就是文件监测。在此或许有很多朋友并不知晓文件监测有何意义及目，那么请大家回想下调试成**热发布**功能后的 Web 容器。当项目迭代后并重新部署时，开发人员无需对其进行手动重启，因为 Web 容器一旦监测到文件发生改变后，便会自动去**适应**这些“变化”并重新进行内部**装载**。Web 容器的热发布功能同样也是基于文件监测功能，所以不得不承认，文件监测功能的出现对于 Java 文件系统来说是具有**重大意义**的。

提示：

就事论事而言，Java7.x 的文件监测功能多少存在一些性能和功能上的**缺陷**。但随着 Java 后续版本的迭代，笔者相信会有那么一天，足以让某些整天在论坛上打口水战的“高手”们闭嘴。

如果在程序中需要使用 Java7.x 的文件监测功能，那么我们务必需要了解 java.nio.file 包下的 WatchService 接口。WatchService 接口不仅作为监测服务，还管理着具体的**监控细节**。

我们可以通过使用 java.nio.file 包下的 FileSystems 类型，并调用 FileSystems 类型的 newWatchService()方法，从而获取到 WatchService 接口的对象实例。

获取 WatchService 接口实例：

Java 代码

```
WatchService watchService = FileSystems.getDefault()
        .newWatchService();
```

文件监测是基于**事件驱动**的，事件触发是作为监测的**先决条件**。开发人员可以使用 java.nio.file 包下的 StandardWatchEventKinds 类型提供的3种字面常量来定义监测事件类型，值得注意的是监测事件需要和 WatchService 实例一起进行**注册**。

StandardWatchEventKinds 类型提供的监测事件：

1、**ENTRY_CREATE：文件或文件夹新建事件；**

2、**ENTRY_DELETE：文件或文件夹删除事件；**

3、**ENTRY_MODIFY：文件或文件夹粘贴事件；**

使用 WatchService 类型实现文件监控完整示例：

Java 代码

```
@Test
public void testWatch() {
    /* 监控目标路径 */
    Path path = Paths.get("C:/");
    try {
        /* 创建文件监控对象 */
        WatchService watchService = FileSystems.getDefault()
                .newWatchService();

        /* 注册文件监控的所有事件类型 */
        path.register(watchService, ENTRY_CREATE, ENTRY_DELETE,
                ENTRY_MODIFY);

        /* 循环监测文件 */
        while (true) {
            WatchKey watchKey = watchService.take();

            /* 迭代触发事件的所有文件 */
            for (WatchEvent<?> event : watchKey.pollEvents())
                System.out.println(event.context().toString() +
" 事件类型: "
                        + event.kind());
            if (!watchKey.reset())
```

```
                return;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

通过上述程序示例我们可以看出，使用 WatchService 接口进行文件监控非常简单和方便。首先我们需要定义好目标监控路径，然后调用 FileSystems 类型的 newWatchService()方法创建 WatchService 对象。接下来我们还需使用 Path 接口的 register()方法注册 WatchService 实例及监控事件。当这些基础作业层全部准备好后，我们再编写外围**实时监测**循环。最后迭代 WatchKey 来获取所有**触发**监控事件的文件即可。

提示：

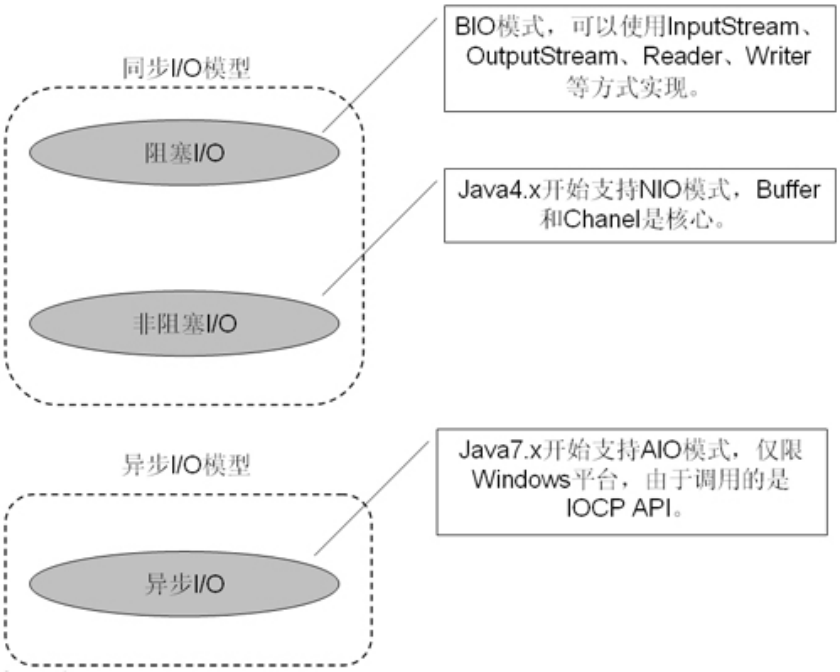
如果在项目中使用上述程序示例，笔者建议你最好将这段监控代码进行**异步化**。因为死循环会占用主线程，后续代码将永远得不到执行机会。

八、探讨 Java I/O 模型

在基于分布式的编程环境中，系统性能的**瓶颈**往往由 I/O 读写性决定。这是不可否认的事实，也是众多开发人员首要考虑的优化问题。如果在 Windows 环境下我们使用阻塞 I/O 模型来编写分布式应用，其维护成本的往往超出你的想象。因为客户端的**链接数量直接决定了服务器内存开辟的线程数量 *2**（包含：接受线程、输出线程），并且这些线程是无法采取线程池优化的，因为线程的执行之间**大于**其创建和销毁时间。长时间的大量并发线程挂起，不仅 CPU 要做实时**任务切换**，其整体物理资源都将一步步被**蚕食**，直至最后程序崩溃。在早期的网络编程中，采取阻塞 I/O 模型来编写分布式应用，唯一能做性能优化只有采取传统的**硬件式堆机**。在付出高昂的硬件成本开销时，其项目的维护性也令开发人员头痛。而且在实际的开发过程中，大部分开发人员会选择将项目部署在 Linux 下运行。跟 Windows **内核结构**不同的是，Linux 环境下是没有真正意义上的线程概念。其所谓的线程都是采用**进程模拟**的方式，也就是**伪线程**。笔者希望大家能够明白，对于并发要求极高的分布式应用，一旦采用阻塞 IO 模型编写就等于自寻死路。

Java 的 I/O 模型由同步 I/O 和异步 I/O 构成。同步 I/O 模型包含：阻塞 I/O 和非阻塞 I/O，而在 Windows 环境下只要调用了 IOCP 的 I/O 模型，就是真正意义上的异步 I/O。

Java 的 I/O 模型示例图：



IOCP（Input/Outut Completion Port，输入/输出完成端口）简单来说是一种**系统级**的**高性能**异步 I/O 模型。应用程序中所有的 I/O 操作将全部**委托**给操作系统线程去执行，直至最后通知并返回结果。Java7.x 对 IOCP 进行了**深度封装**，这使得开发人员可以使用 IOCP API 编写高效的分布式应用。当然 IOCP **仅限**于使用在 Windows 平台，因而无法在 Linux 平台上使用它（Linux 平台上可以通过 Epoll 模拟 IOCP 实现）。

提示：

有过网络编程经验的开发人员都应该明白，在 Windows 平台下性能最好的 I/O 模型是 IOCP，Linux 平台下则是 EPOLL。但是 EPOLL 并不算真正意义上的异步 I/O，EPOLL 只是在尽可能的**模拟** IOCP 而已。因为按照 Unix 网络编程的划分，多路复用 I/O 仍然属于同步 I/O 模型，也就是说 EPOLL 其实是属于多路复用 I/O。

简单来说异步 I/O 的特征必须满足如下2点：

1、I/O 请求与 I/O 操作不会阻塞；

2、并非程序自身完成 I/O 操作，由操作系统线程处理实际的 I/O 操作，直至最后通知并返回结果；

早在 Java4.x 的时候，NIO（Java New Input/Output，Java 新输入/输出）的出现，使得开发人员可以彻底从阻塞 I/O 的噩梦中挣脱出来。但编写 NIO 的成本较大，学习难度也比较高，使得诸多开发人员望而却步（目前比较成熟的 NIO Frameworks 有：Mina、Netty）。但理解非阻塞 I/O 的原理还是非常有必要，先来观察下述采用阻塞 I/O 模式编写的分布式应用示例：

Java 代码

```
@Test
public void testServer() {
    try {
        ServerSocket server = new ServerSocket(8888);
        Socket clist = server.accept();
        BufferedReader reader = new BufferedReader(new
InputStreamReader(
                clist.getInputStream()));

        /* 未收到 I/O 请求时阻塞 */
        System.out.println(reader.readLine());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

I/O 的工作内容我们可以根据其性质划分为2部分：I/O 请求和 I/O 操作。上述程序示例我们采用的是阻塞 I/O 模型，可以很明确的发现当客户端成功握手服务端后，如果服务端并没有收到客户端的 I/O 请求，服务端会在 reader.readLine()方法处阻塞。直到成功接收到 I/O 请求后，服务端才会开始执行实际的 I/O 操作。运用阻塞 I/O 模式进行分布式编程，为了保证服务端与客户端集合的成功会话，我们不得不为每一条客户端连接都开辟独立的线程执行

I/O 操作。当然在实际的开发过程中，或许已经没有开发人员会做这么**荒唐**的事情了。

非阻塞 I/O 和阻塞 I/O 最大的不同在于，非阻塞 I/O 并不会在 I/O 请求时产生阻塞。也就是说如果服务端没有收到 I/O 请求时，非阻塞 I/O 会**“持续轮循”** I/O 请求，当有请求进来后就开始执行 I/O 操作并**阻塞请求进程**。Java7.x 允许开发人员使用 IOCP API 进行异步 I/O 编程，这使得开发人员不必再关心 I/O **是否阻塞**。因为应用程序中所有的 I/O 操作将全部**委托**给操作系统线程去执行，直至最后通知并返回结果。

九、Swing Framework（JSR 296规范）支持

笔者其实对 Swing **非常厌恶**，如果可以的话笔者希望 Oracle 能够**废除**掉 Swing 这项技术。早在08年的时候笔者由于项目需要，曾饱受 Swing 的折磨。**繁琐的布局、组件加载优化、冗长代码维护**等这些令人痛苦和发指的问题，笔者相信使用过 Swing 的人开发人员都能发出相同的感叹。

早期的 Java GUI（图形用户界面）主要由 AWT 技术主导，但随着用户对**胖客户端**技术的丰富度要求逐渐提高，AWT 主键逐渐被 Swing 替代。Swing 其实继承于 AWT，并提供有更加绚丽的视图组件效果。何况相对于**重量级**的 AWT 组件来说，Swing 显得更加轻量。

笔者刚才说过，Swing 虽然相对于 AWT 来说组件内容更加丰富，但仍然掩盖不了其**繁琐**的操作实现。如果对组件性能有过高要求，或者需要实现快速开发，笔者更建议你使用 SWT 或者 JFace 技术(**无需指望使用 IDE 工具进行可视化编程，因为这纯粹是吃力不讨好**)。因为这2种技术可以看成是 Swing 的过渡，且相对 Swing 来说性能和丰富度更加优秀。

既然 Java7.x 对 Swing 仍不忘眷顾优化，那希望大家还是支持一下吧。从官方声明可以看出，JSR 296规范的目标是简化 Swing 的开发难度，且提供有更加丰富的组件资源。如果对于从未接触过 Swing 编程的开发人员，笔者倒是建议你尝试一下，或许你并不反感。

十、JVM 内核升级之 Class Loader 架构调整

类装载器（Class Loader）属于 JVM 体系结构的重要**组成部分**，它是将 Java 类型装载进 JVM 内部的**关键一环**。它使得 Java 类型可以动态的被装载到 JVM 内

部解释并运行。

在 JVM 内部存在着2种类型的类装载器：非自定义类装载器和自定义类装载器 。非自定义类装载器负责装载 Java API 中的类型及 Java 程序中的类型，而自定义类装载器能够使用自定义的方式来装载其类型。不同类型的类装载器所装载的类型将被存放于 JVM 内部不同的**命名空间**中。

非自定义类装载器由 JVM 内部3个核心类装载器构成：

1、BootStrap ClassLoader；

2、ExtClassLoader；

3、AppClassLoader；

BootStrap ClassLoader 也称为**启动类装载器**，它是 JVM 内部最**顶层**的类装载器。BootStrap ClassLoader 主要负责装载核心 Java API 中的类型。ExtClassLoader 负责装载扩展目录下的类型。AppClassLoader 则负责装载 ClassPath（Java 应用类路径）下指定的所有类型。其中 ExtClassLoader 和 AppClassLoader 都属于 BootStrap ClassLoader 的**派生**类装载器。

在 Object 内部封装着一个通过 JNI（Java Native Interface，Java 本地接口）方式调用的 getClass()本地方法，该方法返回一个 Class 类型。对于开发人员而言，允许直接调用其 getClassLoader()方法获取类装载器实例。

使用 getClassLoader()方法获取类装载器：

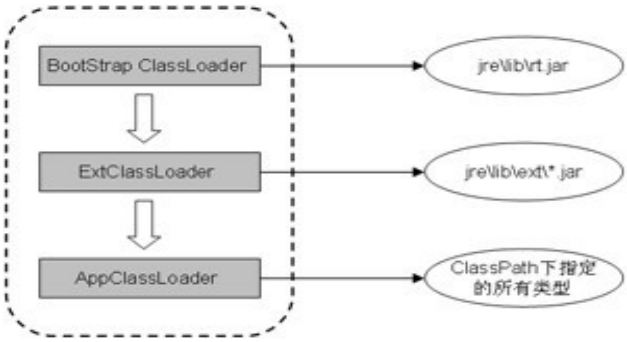
Java 代码

```
@Test
public void testClassLoader() throws Exception {
    /* BootStrap ClassLoader 装载 Java API 中的类型 */
    ClassLoader loader = System.class.getClassLoader();
    System.out.println(null != loader ?
loader.getClass().getName()
            : loader);

    /* ExtClassLoader 装载扩展目录下的类型 */

```

```
System.out.println(CollationData_ar.class.getClassLoader().getClass()
                .getName());

    /* AppClassLoader 装载 ClassPath 下指定的所有类型 */

System.out.println(this.getClass().getClassLoader().getClass()
                .getName());
}
```

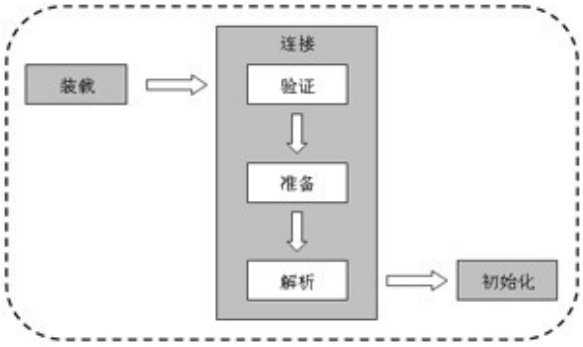
通过上述程序示例我们可以看出，System 类型是被 BootStrap ClassLoader 所装载的。但程序的输出结果却是 Null，当然这并不代表 BootStrap ClassLoader 不存在。因为 BootStrap ClassLoader **并不是采用 Java 语言编写**，而是由 C++ 语言编写并**嵌入**在 JVM 内部，所以开发人员无法获取其实例。CollationData_ar 类型属于 jre\lib\ext 目录下的派生类，由 ExtClassLoader 装载。当前类则由 AppClassLoader 负责装载。在此笔者要提示大家，ExtClassLoader 和 AppClassLoader 都是采用 Java 语言编写。所以 ExtClassLoader 和 AppClassLoader **本身也都是 Java 类型**，都会被最顶层的类装载器 BootStrap ClassLoader 装载，最后才会装载各自管辖范围内的类型。谈到 ClassLoader 的架构，我们不得不提及**双亲委派模型**。在 JVM 内部，类装载器装载类型所采用的便是双亲委派模型机制。比如 AppClassLoader 需要将一个类型装载进 JVM 内部，首先其自身并不会立即装载，而是将目标类型**委派**给上一级，也就是 ExtClassLoader。ExtClassLoader 接着再继续委派给 BootStrap ClassLoader。在 JVM 内部最顶层的类装载器就是 BootStrap ClassLoader，首先由它负责装载目标类型及其关联或依赖的所有类型。如果 BootStrap ClassLoader 装载失败，则退回给 ExtClassLoader 装载。如果 ExtClassLoader 也无法装载，最后只能退回给 AppClassLoader 继续装载。最后当 AppClassLoader 都无法装载的时候，便会抛出 ClassNotFoundException 异常（**开发人员可以在捕获 ClassNotFoundException 异常的时候重写 ClassLoder 类型的 findClass()方法实现自定义类型装载**）。

类装载器架构示例：



类装载器除了需要负责类型的装载，还需要负责验证目标类型的正确性、属性内存分配、解析符号引用等操作。JVM 通过装载、连接和初始化一个 Java 类型，使其可以被运行时的 Java 应用程序所使用。其中装载就是把二进制形式的 Java 类型**写入**进 JVM 内部。连接则是把已经写入进 JVM 中的二进制形式的类型**合并**到 JVM 的运行时状态中去。然而连接阶段又分成了3个步骤：验证、准备和解析。“验证”步骤确保了 Java 类型的数据格式，“准备”步骤则负责为目标类型分配所需的内存空间，“解析”步骤负责把常量池中的符号引用转换为直接使用。“验证”和“解析”这2个步骤都是为最后的初始化工作做准备。

类型生命周期示例：



Java7.x 在上述 ClassLoader 架构的基础之上，进行了一些细微**调整**。在早期开发人员如果想要实现自定义类装载器，恐怕只能实现 ClassLoader 类型并重写其 findClass()方法。但由于 findClass()方法是按照**串行**结构的方式执行，**或许是出于对性能和安全的考虑**。Java7.x 提供了一个拥有并行执行能力的增强实现，这样一来自定义类装载器便可以通过**异步**方式对类型进行装载。

原文链接：http://seanzhou.iteye.com/blog/2002813?utm_source=tuicool

深入理解 JavaScript 定时机制

容易欺骗别人感情的 JavaScript 定时器

JavaScript 的 setTimeout 与 setInterval 是两个很容易欺骗别人感情的方法,因为我们开始常常以为调用了就会按既定的方式执行, 我想不少人都深有同感, 例如

```
1       setTimeout(function() {
2           alert('你好!');
3       }, 0);
4       setInterval(callbackFunction, 100);
```

认为 setTimeout 中的问候方法会立即被执行,因为这并不是凭空而说,而是 JavaScript API 文档明确定义第二个参数意义为隔多少毫秒后,回调方法就会被执行. 这里设成0毫秒,理所当然就立即被执行了.

同理对setInterval 的callbackFunction 方法每间隔100毫秒就立即被执行深信不疑!

但随着 JavaScript 应用开发经验不断的增加和丰富,有一天你发现了一段怪异的代码而百思不得其解:

```
1       div.onclick = function(){
2               setTimeout(function() {
3   document.getElementById('inputField').focus();
4               }, 0);
5       };
```

既然是0毫秒后执行,那么还用 setTimeout 干什么, 此刻, 坚定的信念已开始动摇.

直到最后某一天 , 你不小心写了一段糟糕的代码:

```
1       setTimeout(function() {
2               while (true) {
3               }
4       }, 100);
5       setTimeout(function() {
6               alert('你好!');
7       }, 200);
8       setInterval(callbackFunction, 200);
```

第一行代码进入了死循环,但不久你就会发现,第二,第三行并不是预料中的事情,alert 问候未见出现,callbacKFunction 也杳无音讯!

这时你彻底迷惘了,这种情景是难以接受的,因为改变长久以来既定的认知去接受新思想的过程是痛苦的,但情事实摆在眼前,对 JavaScript 真理的探求并不会因为痛苦而停止,下面让我们来展开 JavaScript 线程和定时器探索之旅!

拔开云雾见月明

出现上面所有误区的最主要一个原因是:潜意识中认为,JavaScript 引擎有多个线程在执行,JavaScript 的定时器回调函数是异步执行的.

而事实上的,JavaScript 使用了障眼法,在多数时候骗过了我们的眼睛,这里背光得澄清一个事实:

JavaScript 引擎是单线程运行的,浏览器无论在什么时候都只且只有一个线程在运行 JavaScript 程序.

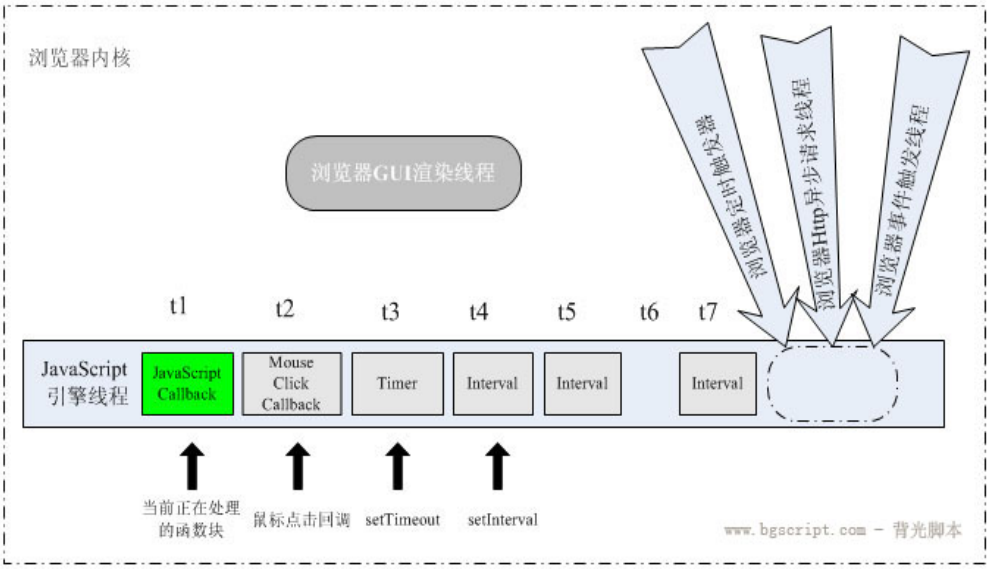
JavaScript 引擎用单线程运行也是有意义的,单线程不必理会线程同步这些复杂的问题,问题得到简化.

那么单线程的 JavaScript 引擎是怎么配合浏览器内核处理这些定时器和响应浏览器事件的呢?

下面结合浏览器内核处理方式简单说明.

浏览器内核实现允许多个线程异步执行,这些线程在内核制控下相互配合以保持同步.假如某一浏览器内核的实现至少有三个常驻线程:javascript 引擎线程,界面渲染线程,浏览器事件触发线程,除些以外,也有一些执行完就终止的线程,如 Http 请求线程,这些异步线程都会产生不同的异步事件,下面通过一个图

来阐明单线程的 JavaScript 引擎与另外那些线程是怎样互动通信的.虽然每个浏览器内核实现细节不同,但这其中的调用原理都是大同小异.



Js 线程图示

由图可看出,浏览器中的 JavaScript 引擎是基于事件驱动的,这里的事件可看作是浏览器派给它的各种任务,这些任务可以源自 JavaScript 引擎当前执行的代码块,如调用 setTimeout 添加一个任务,也可来自浏览器内核的其它线程,如界面元素鼠标点击事件,定时触发器时间到达通知,异步请求状态变更通知等.从代码角度看来任务实体就是各种回调函数,JavaScript 引擎一直等待着任务队列中任务的到来.由于单线程关系,这些任务得进行排队,一个接着一个被引擎处理.

上图 t1-t2..tn 表示不同的时间点,tn 下面对应的小方块代表该时间点的任务,假设现在是 t1时刻,引擎运行在 t1对应的任务方块代码内,在这个时间点内,我们来描述一下浏览器内核其它线程的状态.

t1时刻:

GUI 渲染线程:

该线程负责渲染浏览器界面 HTML 元素,当界面需要重绘(Repaint)或由于某种操作引发回流(reflow)时,该线程就会执行.本文虽然重点解释 JavaScript 定时机制,但这时有必要说说渲染线程,因为该线程与 JavaScript 引擎线程是互斥的,这容易理解,因为 JavaScript 脚本是可操纵 DOM 元素,在修改这些元素属性同时渲染界面,那么渲染线程前后获得的元素数据就可能不一致了.

在 JavaScript 引擎运行脚本期间,浏览器渲染线程都是处于挂起状态的,也就是说被”冻结”了.

所以,在脚本中执行对界面进行更新操作,如添加结点,删除结点或改变结点的外观等更新并不会立即体现出来,这些操作将保存在一个队列中,待 JavaScript 引擎空闲时才有机会渲染出来.

GUI 事件触发线程:

JavaScript 脚本的执行不影响 html 元素事件的触发,在 t1时间段内,首先是用户点击了一个鼠标键,点击被浏览器事件触发线程捕捉后形成一个鼠标点击事件,由图可知,对于 JavaScript 引擎线程来说,这事件是由其它线程异步传到任务队列尾的,由于引擎正在处理 t1时的任务,这个鼠标点击事件正在等待处理.

定时触发线程:

注意这里的浏览器模型定时计数器并不是由 JavaScript 引擎计数的,因为 JavaScript 引擎是单线程的,如果处于阻塞线程状态就计不了时,它必须依赖外部来计时并触发定时,所以队列中的定时事件也是异步事件.

由图可知,在这 t1的时间段内,继鼠标点击事件触发后,先前已设置的 setTimeout 定时也到达了,此刻对 JavaScript 引擎来说,定时触发线程产生了一个异步定时事件并放到任务队列中, 该事件被排到点击事件回调之后,等待处理. 同理, 还是在 t1时间段内,接下来某个 setInterval 定时器也被添加了,由于是间隔定时,在 t1段内连续被触发了两次,这两个事件被排到队尾等待处理.

可见,假如时间段 t1非常长,远大于 setInterval 的定时间隔,那么定时触发线程就会源源不断的产生异步定时事件并放到任务队列尾而不管它们是否已被处理,但一旦 t1和最先的定时事件前面的任务已处理完,这些排列中的定时事件就依次不间断的被执行,这是因为,对于 JavaScript 引擎来说,在处理队列中的各任务处理方式都是一样的,只是处理的次序不同而已.

t1过后,也就是说当前处理的任务已返回,JavaScript 引擎会检查任务队列,发现当前队列非空,就取出 t2下面对应的任务执行,其它时间依此类推,由此看来:

如果队列非空,引擎就从队列头取出一个任务,直到该任务处理完,即返回后引擎接着运行下一个任务,在任务没返回前队列中的其它任务是没法被执行的.

相信您现在已经很清楚 JavaScript 是否可多线程,也了解理解 JavaScript 定时器运行机制了,下面我们来对一些案例进行分析:

案例1:setTimeout 与 setInterval

```
1       setTimeout(function() {
2               /* 代码块... */
3               setTimeout(arguments.callee, 10);
4       }, 10);
5
6       setInterval(function(){
7               /*代码块... */
8       }, 10);
```

这两段代码看一起效果一样,其实非也,第一段中回调函数内的 setTimeout 是 JavaScript 引擎执行后再设置新的 setTimeout 定时, 假定上一个回调处理完到下一个回调开始处理为一个时间间隔,理论两个 setTimeout 回调执行时间间隔>=10ms .第二段自 setInterval 设置定时后,定时触发线程就会源源不断的每隔十秒产生异步定时事件并放到任务队列尾,理论上两个 setInterval 回调执行时间间隔<=10.

案例2:ajax 异步请求是否真的异步?

很多同学朋友搞不清楚,既然说 JavaScript 是单线程运行的,那么 XMLHttpRequest 在连接后是否真的异步?

其实请求确实是异步的,不过这请求是由浏览器新开一个线程请求(参见上图),当请求的状态变更时,如果先前已设置回调,这异步线程就产生状态变更事件放到 JavaScript 引擎的处理队列中等待处理,当任务被处理时,JavaScript 引擎始终是单线程运行回调函数,具体点即还是单线程运行 onreadystatechange 所设置的函数.

原文链接：http://blog.sae.sina.com.cn/archives/2394?utm_source=tuicool

超实用的 JavaScript 技巧及最佳实践

文中所提供的代码片段都已经过最新版的 Chrome 30测试，该浏览器使用 V8 JavaScript 引擎（V8 3.20.17.15）。

1.使用逻辑符号&&或者||进行条件判断

[js] view plaincopy

```
var foo = 10;
foo == 10 && doSomething(); // is the same thing as if (foo == 10) doSomething();
foo == 5 || doSomething(); // is the same thing as if (foo != 5) doSomething();
```

AND 也可以用来设置函数参数的默认值

[js] view plaincopy

```
Function doSomething(arg1){
    Arg1 = arg1 || 10; // arg1 will have 10 as a default value if it’s not already set
}
```

2.使用 map()方法来遍历数组

[js] view plaincopy

```
var squares = [1,2,3,4].map(function (val) {
    return val * val;
});
// squares will be equal to [1, 4, 9, 16]
```

3.舍入小数位数

[js] view plaincopy

```
var num =2.443242342;
num = num.toFixed(4);  // num will be equal to 2.4432
```

4.浮点数问题

[js] view plaincopy

```
0.1 + 0.2 === 0.3 // is false
9007199254740992 + 1 // is equal to 9007199254740992
9007199254740992 + 2 // is equal to 9007199254740994
```

0.1+0.2等于0.30000000000000004，为什么会发生这种情况？根据 IEEE754 标准，你需要知道的是所有 JavaScript 数字在64位二进制内的都表示浮点数。开发者可以使用 toFixed()和 toPrecision()方法来解决这个问题。

5.使用 for-in loop 检查遍历对象属性

下面这段代码主要是为了避免遍历对象属性。

[js] view plaincopy

```
for (var name in object) {
    if (object.hasOwnProperty(name)) {
        // do something with name
    }
}
```

6.逗号操作符

[js] view plaincopy

```
var a = 0;
var b = ( a++, 99 );
console.log(a);  // a will be equal to 1
console.log(b);  // b is equal to 99
```

7.计算或查询缓存变量

在使用 jQuery 选择器的情况下，开发者可以缓存 DOM 元素

[js] view plaincopy

```
var navright = document.querySelector('#right');
var navleft = document.querySelector('#left');
var navup = document.querySelector('#up');
var navdown = document.querySelector('#down');
```

8.在将参数传递到 isFinite()之前进行验证

[js] view plaincopy

```
isFinite(0/0) ; // false
isFinite("foo"); // false
isFinite("10"); // true
isFinite(10);   // true
isFinite(undifined);  // false
isFinite();   // false
isFinite(null);  // true  !!!
```

9.在数组中避免负向索引

```
[js] view plaincopy
var numbersArray = [1,2,3,4,5];
var from = numbersArray.indexOf("foo") ;  // from is equal to -1
numbersArray.splice(from,2);     // will return [5]
```

确保参数传递到 indexOf()方法里是非负向的。

10.（使用 JSON）序列化和反序列化

```
[js] view plaincopy
var person = {name :'Saad', age : 26, department : {ID : 15, name : "R&D"} };
var stringFromPerson = JSON.stringify(person);
/*          stringFromPerson          is          equal          to "{"name":"Saad","age":26,"department":{"ID":15,"name":"R&D"}}"    */
var personFromString = JSON.parse(stringFromPerson);
/* personFromString is equal to person object  */
```

11.避免使用 eval()或 Function 构造函数

eval()和 Function 构造函数被称为脚本引擎，每次执行它们的时候都必须把源码转换成可执行的代码，这是非常昂过的操作。

```
[js] view plaincopy
var func1 = new Function(functionCode);
var func2 = eval(functionCode);
```

12.避免使用 with()方法

如果在全局区域里使用 with()插入变量，那么，万一有一个变量名字和它名字一样，就很容易混淆和重写。

13.避免在数组里使用 for-in loop 而不是这样用：

```
[js] view plaincopy
var sum = 0;
for (var i in arrayNumbers) {
    sum += arrayNumbers[i];
}
```

这样会更好：

```
[js] view plaincopy
var sum = 0;
for (var i = 0, len = arrayNumbers.length; i < len; i++) {
    sum += arrayNumbers[i];
}
```

这样会更快：

```
[js] view plaincopy
for (var i = 0; i < arrayNumbers.length; i++)
```

为什么？数组长度 arraynNumbers 在每次 loop 迭代时都会被重新计算。

14.不要向 setTimeout()和 setInterval()方法里传递字符串

如果在这两个方法里传递字符串，那么字符串会像 eval 那样重新计算，这样速度就会变慢，而不是这样使用：

```
[js] view plaincopy
setInterval('doSomethingPeriodically()', 1000);
setTimeOut('doSomethingAfterFiveSeconds()', 5000);
```

相反，应该这样用：

```
[js] view plaincopy
setInterval(doSomethingPeriodically, 1000);
setTimeOut(doSomethingAfterFiveSeconds, 5000);
```

15.使用 switch/case 语句代替较长的 if/else 语句

如果有超过2个以上的 case，那么使用 switch/case 速度会快很多，而且代码看起来更加优雅。

16. 遇到数值范围时，可以选用 switch/casne

```
[js] view plaincopy
function getCategory(age) {
    var category = "";
    switch (true) {
        case isNaN(age):
            category = "not an age";
            break;
        case (age >= 50):
            category = "Old";
            break;
        case (age <= 20):
            category = "Baby";
            break;
        default:
            category = "Young";
            break;
    };
    return category;
}
getCategory(5);  // will return "Baby"
```

17. 创建一个对象，该对象的属性是一个给定的对象

可以编写一个这样的函数，创建一个对象，该对象属性是一个给定的对象，好比这样：

```
[js] view plaincopy
function clone(object) {
    function OneShotConstructor(){};
```

```
    OneShotConstructor.prototype= object;
    return new OneShotConstructor();
}
clone(Array).prototype ;  // []
```

18.一个 HTML escaper 函数

```
[js] view plaincopy
function escapeHTML(text) {
    var replacements= {"<": "<", ">": ">","&": "&", "\"": """};
    return text.replace(/[<>&"]/g, function(character) {
        return replacements[character];
    });
}
```

19.在一个 loop 里避免使用 try-catch-finally

try-catch-finally 在当前范围里运行时会创建一个新的变量，在执行 catch 时，捕获异常对象会赋值给变量。

不要这样使用：

```
[js] view plaincopy
var object = ['foo', 'bar'], i;
for (i = 0, len = object.length; i <len; i++) {
    try {
        // do something that throws an exception
    }
    catch (e) {
        // handle exception
    }
}
```

应该这样使用：

```
[js] view plaincopy
var object = ['foo', 'bar'], i;
```

```
try {
    for (i = 0, len = object.length; i <len; i++) {
        // do something that throws an exception
    }
}
catch (e) {
    // handle exception
}
```

20.给 XMLHttpRequests 设置 timeouts

如果一个 XHR 需要花费太长时间，你可以终止链接（例如网络问题），通过给 XHR 使用 setTimeout()解决。

```
[js] view plaincopy
var xhr = new XMLHttpRequest ();
xhr.onreadystatechange = function () {
    if (this.readyState == 4) {
        clearTimeout(timeout);
        // do something with response data
    }
}
var timeout = setTimeout( function () {
    xhr.abort(); // call error callback
}, 60*1000 /* timeout after a minute */ );
xhr.open('GET', url, true);
xhr.send();
```

此外，通常你应该完全避免同步 Ajax 调用。

21.处理 WebSocket 超时

一般来说，当创建一个 WebSocket 链接时，服务器可能在闲置30秒后链接超时，在闲置一段时间后，防火墙也可能会链接超时。

为了解决这种超时问题，你可以定期地向服务器发送空信息，在代

码里添加两个函数：一个函数用来保持链接一直是活的，另一个用来取消链接是活的，使用这种方法，你将控制超时问题。

添加一个 timeID……

```
[js] view plaincopy
var timerID = 0;
function keepAlive() {
    var timeout = 15000;
    if (webSocket.readyState == webSocket.OPEN) {
        webSocket.send('');
    }
    timerId = setTimeout(keepAlive, timeout);
}
function cancelKeepAlive() {
    if (timerId) {
        cancelTimeout(timerId);
    }
```

keepAlive()方法应该添加在 WebSocket 链接方法 onOpen()的末端，cancelKeepAlive()方法放在 onClose()方法下面。

22.记住，最原始的操作要比函数调用快

对于简单的任务，最好使用基本操作方式来实现，而不是使用函数调用实现。

例如

```
[js] view plaincopy
var min = Math.min(a,b);
A.push(v);
```

基本操作方式：

```
[js] view plaincopy
var min = a < b ? a b;
A[A.length] = v;
```

原文链接：http://www.html5cn.org/portal.php?mod=view&aid=5848&utm_source=tuicool

红皮书（9）：DOM

Node 类型

nodeType

以下是一些重要的 nodeType 的取值：

1： 元素 element

2： 属性 attr

3： 文本 text

8： 注释 comments

9： 文档 document

nodeName,nodeValue

节点关系

childNodes: 每个节点都有一个 childNodes 属性，其中保存着一个 NodeList 对象

firstChild: 等同于 childNodes[0]

lastChild: 等同于 childNodes.length-1

同时通过使用列表中每个节点的 previousSibling 和 nextSibling 属性，可以访问同一列表中的其他节点。

操作节点

appendChild()

appendChild()方法用于向 childNodes 列表的末尾添加一个节点。添加节点后，childNodes 的新增节点、父节点及以前的最后一个子节点的关系指针都会相应地得到更新。

insertBefore()

insertBefore()这个方法接受两个参数：要插入的节点和作为参照的节点。

```
// 插入后成为最后一个子节点
returnedNode = someNode.insertBefore(newNode,null);
// 插入后成为第一个节点
returnedNode = someNode.insertBefore(newNode,someNode.firstChild);
```

```
// 插入到最后一个子节点前面
returnedNode = someNode.insertBefore(newNode,someNode.lastChild);
```

repaceChild()

repaceChild()接受两个参数，要插入的节点和要替换的节点

```
var returnedNode = someNode.replaceChild(newNode,someNode.firstChild);
```

removeChild()

只移除而非替换节点。

```
var formerFirstChild = someNode.removeChild(someNode.firstChild);
```

cloneNode()

```
• item 1
• item 2
• item 3
var deepList = myList.cloneNode(true);
console.log(deepList.length); // 3
var shallowList = myList.cloneNode(false);
console.log(shallowList.childNodes.length); //0
```

原文链接：http://blog.segmentfault.com/joanna123/1190000000383338?utm_source=tuicool

理解响应式布局设计

讲到响应式布局，相信大家都有一定的了解，响应式布局是今年很流行的一个设计理念，随着移动互联网的盛行，为解决如今各式各样的浏览器分辨率以及不同移动设备的显示效果， 设计师提出了响应式布局的设计方案。今天就和大家分享一个响应式布局，包含什么是响应式布局、响应式布局的优点和缺点以及响应式布局该怎么设计（通过 CSS3 Media Query 实现响应布局）。

什么是响应式布局？

响应式布局是 EthanMarcotte 在2010年5月份提出的一个概念，简而言之，就是一个网站能够兼容多个终端——而不是为每个终端做一个特定的版本。这个概念是为解决移动互联网浏览而诞生的。

响应式布局可以为不同终端的用户提供更加舒适的界面和更好的用户体验，而且随着目前大屏幕移动设备的普及，用大势所趋来形容也不为过。随着越来越多的设计师采用这个技术，我们不仅看到很多的创新，还看到了一些成形的模式。

响应式布局有哪些优点和缺点？

优点：

面对不同分辨率设备灵活性强

能够快捷解决多设备显示适应问题

缺点：

兼容各种设备工作量大，效率低下

代码累赘，会出现隐藏无用的元素，加载时间加长

其实这是一种折衷性质的设计解决方案，多方面因素影响而达不到最佳效果

一定程度上改变了网站原有的布局结构，会出现用户混淆的情况

响式布局该怎么设计？

我们在上面了解了什么是响应式布局，那在我们的实际项目中应该怎么去设计呢？在以往我们设计网站的时候都会受到不同浏览器的兼容性的困扰，现在还要来个不同尺寸设备，我们该怎么淡定下来呢？有需求就会有解决方案，呵呵，说到响应式布局，就不得不提起 CSS3中的 Media Query（媒介查询），这可是个好东西，易用、强大、快捷……Media Query 是制作响应式布局的一个利器，使用这个工具，我们可以非常方便快捷的制造出各种丰富的实用性强的界面。接下来就一起来深入的了解 Media Query。

1、CSS 中的 Media Query（媒介查询）是什么？

通过不同的媒体类型和条件定义样式表规则。媒体查询让 CSS 可以更精确作用于不同的媒体类型和同一媒体的不同条件。媒体查询的大部分媒体特性都接受 min 和 max 用于表达”大于或等于”和”小与或等于”。如：width 会有 min-width 和 max-width 媒体查询可以被用在 CSS 中的@media 和@import 规则上，也可以被

用在 HTML 和 XML 中。通过这个标签属性，我们可以很方便的在不同的设备下实现丰富的界面，特别是移动设备，将会运用更加的广泛。

2、media query 能够获取哪些值？

设备的宽和高 device-width，device-heigth 显示屏幕/触觉设备。

渲染窗口的宽和高 width，heigth 显示屏幕/触觉设备。

设备的手持方向，横向还是竖向 orientation（portrait|lanscape）和打印机等。

画面比例 aspect-ratio 点阵打印机等。

设备比例 device-aspect-ratio-点阵打印机等。

对象颜色或颜色列表 color，color-index 显示屏幕。

设备的分辨率 resolution。

3、语法结构及用法

1

@media 设备名 only （选取条件）not （选取条件）and（设备选取条件），设备二{sRules}

示例一：在 link 中使用@media：

1

<link rel=“stylesheet” type=“text/css” media=“only screen and （max-width：480px），only screen and （max-device-width：480px）” href=“link.css” />

上面使用中 only 可省略，限定于计算机显示器，第一个条件 max-width 是指渲染界面最大宽度，第二个条件 max-device-width 是指设备最大宽度。

示例二：在样式表中内嵌@media：

1

@media （min-device-width:1024px）and （max-width:989px），screen and （max-device-width:480px），（max-device-width:480px） and （orientation:landscape），（min-device-width:480px） and （max-device-width:1024px） and （orientation:portrait） {srules}

在示例二中，设置了电脑显示器分辨率（宽度）大于或等于1024px（并且最

大可见宽度为989px）；屏宽在480px 及其以下手持设备；屏宽在480px 以及横向（即480尺寸平行于地面）放置的手持设备；屏宽大于或等于480px 小于1024px 以及垂直放置设备的 css 样式。

从上面的例子可以看出，字符间以空格相连，选取条件包含在小括号内，srules 为兼容设置的样式表，包含在中括号里面。only（限定某种设备，可省略），and（逻辑与），not（排除某种设备）为逻辑关键字，多种设备用逗号分隔，这一点继承了 css 基本语法。

4、可用设备名参数：

5、逻辑关键字：

7、测试 Media Queries

最后，我们需要对我们刚刚设计的 Media Queries 进行测试，想要在不同设备上测试 Media Queries 的效果，可以使用一个浏览工具来检验不同尺寸屏幕下的显示效果，在这里为大家介绍一个不错的在线工具 - Responsivator，它可以模拟 iPhone 等各种不同设备，并且还可以自定义不同尺寸屏幕的显示效果，只需要输入一个 url 甚至是本地的一个 url（如：http://127.0.0.1/），就可以看到网站在不同尺寸屏幕下的显示效果。

8、通过 Media Queries 实现响应式布局设计

好了，我们明白了什么是 Media Query，那我们一起来运用到响应式布局的设计项目中去。设计思路很简单，首先先定义在标准浏览器下的固定宽度（假如标准浏览器的分辨率为1024px，那么我们设置宽为980px），然后用 Media Query 来监测浏览器的尺寸变化，当浏览器的分辨率小于1024px 的时候，则通过 Media Query 预设的样式表来将页面的宽度设置为百分比显示，这样子页面的结构元素就会根据浏览器的的尺寸来进行相对应的调整。同理，当浏览器的可视区域改变到某个值（假如为650px）的时候，页面的结构元素根据 Media Query 预设的层叠样式表来进行相对应的调整。看看我们的例子：

1

2

3

4

```
5
6
7
8
9
10
11
12
13
14
/* 当浏览器的可视区域小于980px */
@media screen and （max-width：980px）{
#wrap {width：90%; margin:0 auto;}
#content {width：60%;padding：5%;}
#sidebar {width：30%;}
#footer {padding：8% 5%;margin-bottom：10px;}
}
/* 当浏览器的可视区域小于650px */
@media screen and （max-width：650px）{
#header {height：auto;}
#searchform {position：absolute;top：5px;right：0;}
#content {width：auto; float：none; margin：20px 0;}
#sidebar {width：100%; float：none; margin：0;}
}
```

通过上面我们就可以监测浏览器的可视区域变化的是时候我们的页面结构元素也会相对应的变化，当然你可以再多设置几个尺寸的监测层叠样式表，这样子就可以根据不同尺寸设备来进行响应式的布局。为了更好的显示效果，我们往往还要格式化一些 CSS 属性的初始值：

```
1
```

```
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
/* 禁用 iPhone 中 Safari 的字号自动调整*/
html {
-webkit-text-size-adjust: none;
}
/* 设置 HTML5元素为块*/
article,aside,details,figcaption,figure,footer,header,hgroup,
menu, nav, section {
display: block;
}
/* 设置图片视频等自适应调整*/
img {
max-width: 100%;
height: auto;
width: auto\9; /* ie8 */
}
.video embed, .video object, .video iframe {
width: 100%;
height: auto;
```

}

最后要注意的是在页面的头部<head></head>之间加上下面这句：

1

<meta name=“viewport” content=“width=device-width; initial-scale=1.0” >

metaviewport 这个属性是在移动设备上设置原始大小显示和是否缩放的声明。

参数设置：

width - viewport 的宽度

height - viewport 的高度

initial-scale - 初始的缩放比例

minimum-scale - 允许用户缩放到的最小比例

maximum-scale - 允许用户缩放到的最大比例

user-scalable - 用户是否可以手动缩放

最后对于在 IE 浏览器中不支持 media query 的情况，我们可以使用 Media Query JavaScript 来解决，只需要在页面头部引用 css3-mediaqueries.js 即可。示例：

1

2

3

<! -- [if lt IE 9] >

<scriptsrc= “http://css3-mediaqueries-js.googlecode.com/svn/trunk/css3-mediaqueries.js” ></script>

<! [endif] -->

原文链接：

http://www.cnblogs.com/wqsbk/p/3512732.html?utm_source=tuicool

[编程语言]

(译)KVO 的内部实现

09年的一篇文章，比较深入地阐述了 KVO 的内部实现。

KVO 是实现 Cocoa Bindings 的基础，它提供了一种方法，当某个属性改变时，相应的 objects 会被通知到。在其他语言中，这种观察者模式通常需要单独实现，而在 Objective-C 中，通常无须增加额外代码即可使用。

概览

这是怎么实现的呢？其实这都是通过 Objective-C 强大的运行时(runtime)实现的。当你第一次观察某个 object 时，runtime 会创建一个新的继承原先 class 的 subclass。在这个新的 class 中，它重写了所有被观察的 key，然后将 object 的 isa 指针指向新创建的 class(这个指针告诉 Objective-C 运行时某个 object 到底是哪种类型的 object)。所以 object 神奇地变成了新的子类的实例。

这些被重写的方法实现了如何通知观察者们。当改变一个 key 时，会触发 setKey 方法，但这个方法被重写了，并且在内部添加了发送通知机制。(当然也可以不走 setXXX 方法，比如直接修改 iVar，但不推荐这么做)。

有意思的是：苹果不希望这个机制暴露在外部。除了 setters，这个动态生成的子类同时也重写了-class 方法，依旧返回原先的 class！如果不仔细看的话，被 KVO 过的 object 看起来和原先的 object 没什么两样。

深入探究

下面来看看这些是如何实现的。我写了个程序来演示隐藏在 KVO 背后的机制。

```
// gcc -o kvoexplorer -framework Foundation kvoexplorer.m

#import <Foundation/Foundation.h>
#import <objc/runtime.h>


@interface TestClass : NSObject
{
```

```
    int x;
    int y;
    int z;
}
@property int x;
@property int y;
@property int z;
@end

@implementation TestClass
@synthesize x, y, z;
@end

static NSArray *ClassMethodNames(Class c)
{
    NSMutableArray *array = [NSMutableArray array];

    unsigned int methodCount = 0;
    Method *methodList = class_copyMethodList(c, &methodCount);
    unsigned int i;
    for(i = 0; i < methodCount; i++)
        [array addObject:
NSStringFromSelector(method_getName(methodList[i]))];
    free(methodList);

    return array;
}

static void PrintDescription(NSString *name, id obj)
```

```
{
    NSString *str = [NSString stringWithFormat:
        @"%@:      %@\n\tNSObject      class      %s\n\tlibobjc class %s\n\timplements methods <%@>",
        name,
        obj,
        class_getName([obj class]),
        class_getName(obj->isa),
        [ClassMethodNames(obj->isa) componentsJoinedByString:@", "]];
    printf("%s\n", [str UTF8String]);
}

int main(int argc, char **argv)
{
    [NSAutoreleasePool new];

    TestClass *x = [[TestClass alloc] init];
    TestClass *y = [[TestClass alloc] init];
    TestClass *xy = [[TestClass alloc] init];
    TestClass *control = [[TestClass alloc] init];

    [x addObserver:x forKeyPath:@"x" options:0 context:NULL];
    [xy addObserver:xy forKeyPath:@"x" options:0 context:NULL];
    [y addObserver:y forKeyPath:@"y" options:0 context:NULL];
    [xy addObserver:xy forKeyPath:@"y" options:0 context:NULL];

    PrintDescription(@"control", control);
    PrintDescription(@"x", x);
```

```
    PrintDescription(@"y", y);
    PrintDescription(@"xy", xy);

    printf("Using NSObject methods, normal setX: is %p, overridden setX: is %p\n",
            [control methodForSelector:@selector(setX:)],
            [x methodForSelector:@selector(setX:)]);
    printf("Using libobjc functions, normal setX: is %p, overridden setX: is %p\n",
method_getImplementation(class_getInstanceMethod(object_getClass(control),
                                        @selector(setX:))),
method_getImplementation(class_getInstanceMethod(object_getClass(x),
                                        @selector(setX:))));

    return 0;
}
```

我们从头到尾细细看来。

首先定义了一个 TestClass 的类，它有3个属性。

然后定义了一些方便调试的方法。ClassMethodNames 使用 Objective-C 运行时方法来遍历一个 class，得到方法列表。注意，这些方法不包括父类的方法。PrintDescription 打印 object 的所有信息，包括 class 信息（包括-class 和通过运行时得到的 class），以及这个 class 实现的方法。

然后创建了4个 TestClass 实例，每一个都使用了不同的观察方式。x 实例有一个观察者观察 xkey，y，xy 也类似。为了做比较，zkey 没有观察者。最后 control 实例没有任何观察者。

然后打印出4个 objects 的 description。

之后继续打印被重写的 setter 内存地址，以及未被重写的 setter 的内存地址做比较。这里做了两次，是因为-methodForSelector:没能得到重写的方法。KVO 试图掩盖它实际上创建了一个新的 subclass 这个事实！但是使用运行时的方法就原形毕露了。

运行代码

看看这段代码的输出

```
• control: <TestClass: 0x104b20>
•     NSObject class TestClass
•     libobjc class TestClass
•     implements methods <setX:, x, setY:, y, setZ:, z>
• x: <TestClass: 0x103280>
•     NSObject class TestClass
•     libobjc class NSKVONotifying_TestClass
•     implements methods <setY:, setX:, class, dealloc, _isKVOA>
• y: <TestClass: 0x104b00>
•     NSObject class TestClass
•     libobjc class NSKVONotifying_TestClass
•     implements methods <setY:, setX:, class, dealloc, _isKVOA>
• xy: <TestClass: 0x104b10>
•     NSObject class TestClass
•     libobjc class NSKVONotifying_TestClass
•     implements methods <setY:, setX:, class, dealloc, _isKVOA>
• Using NSObject methods, normal setX: is 0x195e, overridden setX: is 0x195e
• Using libobjc functions, normal setX: is 0x195e, overridden setX: is 0x96a1a550
```

首先，它输出了 controlobject，没有任何问题，它的 class 是 TestClass，并且实现了6个 set/get 方法。

然后是3个被观察的 objects。注意-class 仍然显示的是 TestClass，使用 object_getClass 显示了这个 object 的真面目：它是 NSKVONotifying_TestClass 的一个实例。这个 NSKVONotifying_TestClass 就是动态生成的 subclass！

注意，它是如何实现这两个被观察的 setters 的。你会发现，它很聪明，没有重写-setZ:，虽然它也是个 setter，因为它没有被观察。同时注意到，3个实例对应的是同一个 class，也就是说两个 setters 都被重写了，尽管其中的两个实例只观察了一个属性。这会带来一点效率上的问题，因为即使没有被观察的 property 也会走被重写的 setter，但苹果显然觉得这比分开生成动态的 subclass 更好，我也觉得这是个正确的选择。

你会看到3个其他的方法。有之前提到过的被重写的-class 方法，假装自己还是原来的 class。还有-dealloc 方法处理一些收尾工作。还有一个_isKVOA 方法，看起来像是一个私有方法。

接下来，我们输出-setX:的实现。使用-methodForSelector:返回的是相同的值。因为-setX:已经在子类被重写了，这也就意味着 methodForSelector:在内部实现中使用了-class，于是得到了错误的结果。

最后我们通过运行时得到了不同的输出结果。

作为一个优秀的探索者，我们进入 debugger 来看看这第二个方法的实现到底是怎样的：

```
(gdb) print (IMP)0x96a1a550
$1 = (IMP) 0x96a1a550 <_NSSetIntValueAndNotify>
```

看起来是一个内部方法，对 Foundation 使用 nm -a 得到一个完整的私有方法列表：

```
0013df80 t __NSSetBoolValueAndNotify
```

- 000a0480 t __NSSetCharValueAndNotify
- 0013e120 t __NSSetDoubleValueAndNotify
- 0013e1f0 t __NSSetFloatValueAndNotify
- 000e3550 t __NSSetIntValueAndNotify
- 0013e390 t __NSSetLongLongValueAndNotify
- 0013e2c0 t __NSSetLongValueAndNotify
- 00089df0 t __NSSetObjectValueAndNotify
- 0013e6f0 t __NSSetPointValueAndNotify
- 0013e7d0 t __NSSetRangeValueAndNotify
- 0013e8b0 t __NSSetRectValueAndNotify
- 0013e550 t __NSSetShortValueAndNotify
- 0008ab20 t __NSSetSizeValueAndNotify
- 0013e050 t __NSSetUnsignedCharValueAndNotify
- 0009fcd0 t __NSSetUnsignedIntValueAndNotify
- 0013e470 t __NSSetUnsignedLongLongValueAndNotify
- 0009fc00 t __NSSetUnsignedLongValueAndNotify
- 0013e620 t __NSSetUnsignedShortValueAndNotify

这个列表也能发现一些有趣的东西。比如苹果为每一种 primitive type 都写了对应的实现。Objective-C 的 object 会用到的其实只有__NSSetObjectValueAndNotify，但需要一整套来对应剩下的，而且看起来也没有实现完全，比如 long dobule 或_Bool 都没有。甚至没有为通用指针类型(generic pointer type)提供方法。所以，不在这个方法列表里的属性其实是不支持 KVO 的。

KVO 是一个很强大的工具，有时候过于强大了，尤其是有了自动触发通知机制。现在你知道它内部是怎么实现的了，这些知识或许能帮助你更好地使用它，或在它出错时更方便调试。

原文链接：
http://www.cocoachina.com/applenews/devnews/2014/0107/7667.html?utm_source=tuicool

Java NIO 与 IO 的区别和比较

导读

J2SE1.4以上版本中发布了全新的I/O类库。本文将通过一些实例来简单介绍NIO库提供的一些新特性：非阻塞 I/O，字符转换，缓冲以及通道。

一. 介绍 NIO

NIO 包（java.nio.*）引入了四个关键的抽象数据类型，它们共同解决传统的 I/O 类中的一些问题。

1. Buffer：它是包含数据且用于读写的线形表结构。其中还提供了一个特殊类用于内存映射文件的 I/O 操作。

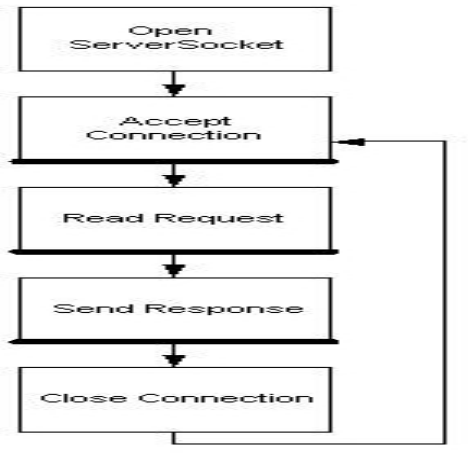
2. Charset：它提供 Unicode 字符串影射到字节序列以及逆影射的操作。

3. Channels：包含 socket，file 和 pipe 三种管道，它实际上是双向交流的通道。

4. Selector：它将多元异步 I/O 操作集中到一个或多个线程中（它可以被看成是 Unix 中 select（）函数或 Win32中 WaitForSingleEvent（）函数的面向对象版本）。

二. 回顾传统

在介绍 NIO 之前，有必要了解传统的 I/O 操作的方式。以网络应用为例，传统方式需要监听一个 ServerSocket，接受请求的连接为其提供服务（服务通常包括了处理请求并发送响应）图一是服务器的生命周期图，其中标有粗黑线条的部分表明会发生 I/O 阻塞。



图一

可以分析创建服务器的每个具体步骤。首先创建 ServerSocket

```
ServerSocket server=new ServerSocket (10000);
```

然后接受新的连接请求

```
Socket newConnection=server.accept ();
```

对于 accept 方法的调用将造成阻塞，直到 ServerSocket 接受到一个连接请求为止。一旦连接请求被接受，服务器可以读客户 socket 中的请求。

```
1       InputStream in = newConnection.getInputStream();
2       InputStreamReader reader = new InputStreamReader(in);
3       BufferedReader buffer = new BufferedReader(reader);
4       Request request = new Request();
5       while(!request.isComplete()) {
6       String line = buffer.readLine();
7       request.addLine(line);
8       }
```

这样的操作有两个问题，首先 BufferedReader 类的 readLine（）方法在其缓冲区未满时会造成线程阻塞，只有一定数据填满了缓冲区或者客户关闭了套接字，方法才会返回。其次，它回产生大量的垃圾，BufferedReader 创建了缓冲区来从客户套接字读入数据，但是同样创建了一些字符串存储这些数据。虽然 BufferedReader 内部提供了 StringBuffer 处理这一问题，但是所有的 String 很快变成了垃圾需要回收。

同样的问题在发送响应代码中也存在

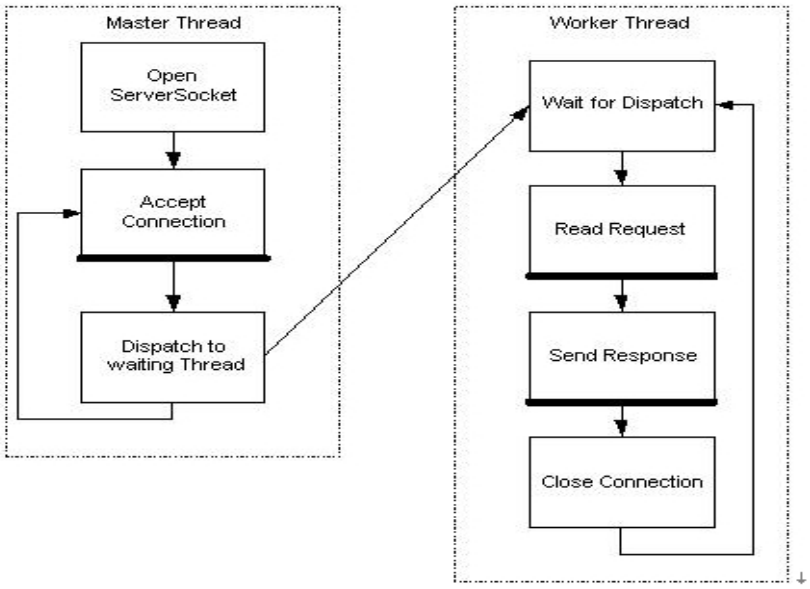
```
1       Response response = request.generateResponse();
2       OutputStream out = newConnection.getOutputStream();
3       InputStream in = response.getInputStream();
4       int ch;
5       while(-1 != (ch = in.read())) {
6       out.write(ch);
```

```
7        }
8        newConnection.close();
```

类似的，读写操作被阻塞而且向流中一次写入一个字符会造成效率低下，所以应该使用缓冲区，但是一旦使用缓冲，流又会产生更多的垃圾。

传统的解决方法

通常在 Java 中处理阻塞 I/O 要用到线程（大量的线程）。一般是实现一个线程池用来处理请求，如图二



图二

线程使得服务器可以处理多个连接，但是它们也同样引发了许多问题。每个线程拥有自己的栈空间并且占用一些 CPU 时间，耗费很大，而且很多时间是浪费在阻塞的 I/O 操作上，没有有效的利用 CPU。

三. 新 I/O

1. Buffer

传统的 I/O 不断的浪费对象资源（通常是 String）。新 I/O 通过使用 Buffer 读写数据避免了资源浪费。Buffer 对象是线性的，有序的数据集合，它根据其类别只包含唯一的数据类型。

java.nio.Buffer 类描述

java.nio.ByteBuffer 包含字节类型。 可以从 ReadableByteChannel 中读在 WritableByteChannel 中写

java.nio.MappedByteBuffer 包含字节类型，直接在内存某一区域映射

java.nio.CharBuffer 包含字符类型，不能写入通道

java.nio.DoubleBuffer 包含 double 类型，不能写入通道

java.nio.FloatBuffer 包含 float 类型

java.nio.IntBuffer 包含 int 类型

java.nio.LongBuffer 包含 long 类型

java.nio.ShortBuffer 包含 short 类型

可以通过调用 allocate(int capacity)方法或者 allocateDirect(int capacity)方法分配一个 Buffer。特别的，你可以创建 MappedBytesBuffer 通过调用 FileChannel.map(int mode,long position,int size)。直接（direct）buffer 在内存中分配一段连续的块并使用本地访问方法读写数据。非直接(nondirect)buffer 通过使用 Java 中的数组访问代码读写数据。有时候必须使用非直接缓冲例如使用任何的 wrap 方法(如 ByteBuffer.wrap(byte[]))在 Java 数组基础上创建 buffer。

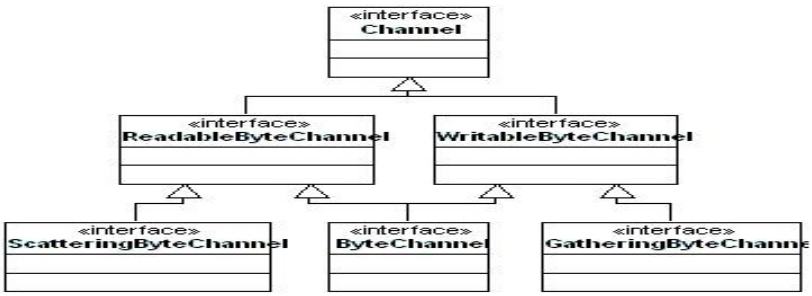
2. 字符编码

向 ByteBuffer 中存放数据涉及到两个问题:字节的顺序和字符转换。ByteBuffer 内部通过 ByteOrder 类处理了字节顺序问题,但是并没有处理字符转换。事实上，ByteBuffer 没有提供方法读写 String。

Java.nio.charset.Charset 处理了字符转换问题。它通过构造 CharsetEncoder 和 CharsetDecoder 将字符序列转换成字节和逆转换。

3. 通道(Channel)

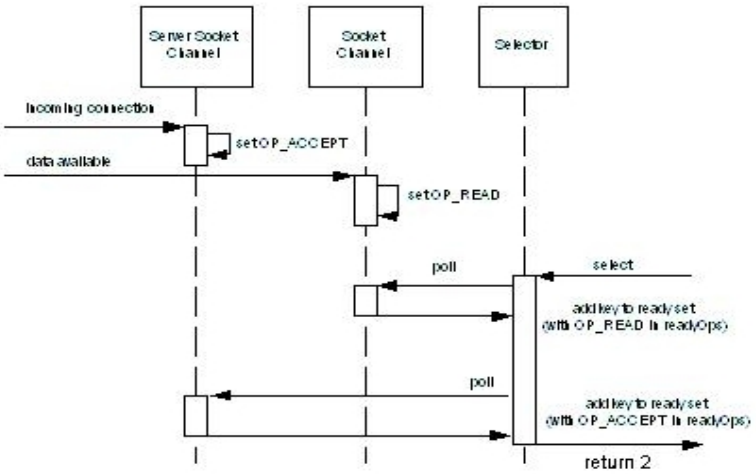
你可能注意到现有的 java.io 类中没有一个能够读写 Buffer 类型，所以 NIO 中提供了 Channel 类来读写 Buffer。通道可以认为是一种连接，可以是到特定设备，程序或者是网络的连接。通道的类等级结构图如下



图中 ReadableByteChannel 和 WritableByteChannel 分别用于读写。GatheringByteChannel 可以从使用一次将多个 Buffer 中的数据写入通道，相反的，ScatteringByteChannel 则可以一次将数据从通道读入多个 Buffer 中。你还可以设置通道使其为阻塞或非阻塞 I/O 操作服务。为了使通道能够同传统 I/O 类相容，Channel 类提供了静态方法创建 Stream 或 Reader

4. Selector

在过去的阻塞 I/O 中，我们一般知道什么时候可以向 stream 中读或写，因为方法调用直到 stream 准备好时返回。但是使用非阻塞通道，我们需要一些方法来知道什么时候通道准备好了。在 NIO 包中，设计 Selector 就是为了这个目的。SelectableChannel 可以注册特定的事件，而不是在事件发生时通知应用，通道跟踪事件。然后，当应用调用 Selector 上的任意一个 selection 方法时，它查看注册了的通道看是否有任何感兴趣的事件发生。图四是 selector 和两个已注册的通道的例子



并不是所有的通道都支持所有的操作。SelectionKey 类定义了所有可能的操作位，将要用两次。首先，当应用调用 SelectableChannel.register(Selector sel, int op)方法注册通道时，它将所需操作作为第二个参数传递到方法中。然后，一旦 SelectionKey 被选中了，SelectionKey 的 readyOps()方法返回所有通道支持操作的数位的和。SelectableChannel 的 validOps 方法返回每个通道允许的操作。注册通道不支持的操作将引发 IllegalArgumentException 异常。下表列出了 SelectableChannel 子类所支持的操作。

1	ServerSocketChannel	OP_ACCEPT

```
2        SocketChannel OP_CONNECT, OP_READ, OP_WRITE
3        DatagramChannel OP_READ, OP_WRITE
4        Pipe.SourceChannel OP_READ
5        Pipe.SinkChannel OP_WRITE
```

四. 举例说明

1. 简单网页内容下载

这个例子非常简单，类 SocketChannelReader 使用 SocketChannel 来下载特定网页的 HTML 内容。

```
package examples.nio;
1       import java.nio.ByteBuffer;
2       import java.nio.channels.SocketChannel;
3       import java.nio.charset.Charset;
4       import java.net.InetSocketAddress;
5       import java.io.IOException;
6       public class SocketChannelReader{
7
8       private Charset charset=Charset.forName("UTF-8");//创
9   建 UTF-8字符集
10      private SocketChannel channel;
11      public void getHTMLContent(){
12      try{
13      connect();
14      sendRequest();
15      readResponse();
16      }catch(IOException e){
17      System.err.println(e.toString());
18      }finally{
19      if(channel!=null){
20      try{
```

```
21      channel.close();
22      }catch(IOException e){}
23      }
24      }
25      }
26      private void connect()throws IOException{//连接到 CSDN
27      InetSocketAddress socketAddress=
28      new InetSocketAddress("http://www.csdn.net",80/);
29      channel=SocketChannel.open(socketAddress);
30      //使用工厂方法 open 创建一个 channel 并将它连接到指定地
31  址上
32      //相当与 SocketChannel.open().connect(socketAddress);
33  调用
34      }
35      private void sendRequest()throws IOException{
36      channel.write(charset.encode("GET "
37      +"/document"
38      +"\r\n\r\n"));//发送 GET 请求到 CSDN 的文档中心
39      //使用 channel.write 方法，它需要 CharByte 类型的参数，
40  使用
41      //Charset.encode(String)方法转换字符串。
42      }
43      private void readResponse()throws IOException{//读取应
44  答
45      ByteBuffer buffer=ByteBuffer.allocate(1024);//创建
46  1024字节的缓冲
47      while(channel.read(buffer)!=-1){
48      buffer.flip();//flip 方法在读缓冲区字节操作之前调用。
49      System.out.println(charset.decode(buffer));
```

```
50      //使用 Charset.decode 方法将字节转换为字符串
        buffer.clear();//清空缓冲
        }
        }
        public static void main(String [] args){
        new SocketChannelReader().getHTMLContent();
        }
```

2. 简单的加法服务器和客户机

服务器代码

```
1       package examples.nio;
2       import java.nio.ByteBuffer;
3       import java.nio.IntBuffer;
4       import java.nio.channels.ServerSocketChannel;
5       import java.nio.channels.SocketChannel;
6       import java.net.InetSocketAddress;
7       import java.io.IOException;
8       /**
9       * SumServer.java
10      *
11      *
12      * Created: Thu Nov 06 11:41:52 2003
13      *
14      * @author starchu1981
15      * @version 1.0
16      */
17      public class SumServer {
18      private ByteBuffer _buffer=ByteBuffer.allocate(8);
```

```
19      private IntBuffer _intBuffer=_buffer.asIntBuffer();
20      private SocketChannel _clientChannel=null;
21      private ServerSocketChannel _serverChannel=null;
22      public void start(){
23      try{
24      openChannel();
25      waitForConnection();
26      }catch(IOException e){
27      System.err.println(e.toString());
28      }
29      }
30      private void openChannel()throws IOException{
31      _serverChannel=ServerSocketChannel.open();
32      _serverChannel.socket().bind(new
33  InetSocketAddress(10000));
34      System.out.println("服务器通道已经打开");
35      }
36      private void waitForConnection()throws IOException{
37      while(true){
38      _clientChannel=_serverChannel.accept();
39      if(_clientChannel!=null){
40      System.out.println("新的连接加入");
41      processRequest();
42      _clientChannel.close();
43      }
44      }
45      }
46      private void processRequest()throws IOException{
47      _buffer.clear();
```

```
48      _clientChannel.read(_buffer);
49      int result=_intBuffer.get(0)+_intBuffer.get(1);
50      _buffer.flip();
51      _buffer.clear();
52      _intBuffer.put(0,result);
53      _clientChannel.write(_buffer);
54      }
55      public static void main(String [] args){
56      new SumServer().start();
57      }
58      } // SumServer
59      客户代码
60      package examples.nio;
61      import java.nio.ByteBuffer;
62      import java.nio.IntBuffer;
63      import java.nio.channels.SocketChannel;
64      import java.net.InetSocketAddress;
65      import java.io.IOException;
66      /**
67      * SumClient.java
68      *
69      *
70      * Created: Thu Nov 06 11:26:06 2003
71      *
72      * @author starchu1981
73      * @version 1.0
74      */
75      public class SumClient {
76      private ByteBuffer _buffer=ByteBuffer.allocate(8);
```

```
77      private IntBuffer _intBuffer;
78      private SocketChannel _channel;
79      public SumClient() {
80      _intBuffer=_buffer.asIntBuffer();
81      } // SumClient constructor
82
83      public int getSum(int first,int second){
84      int result=0;
85      try{
86      _channel=connect();
87      sendSumRequest(first,second);
88      result=receiveResponse();
89      }catch(IOException
90  e){System.err.println(e.toString());
91      }finally{
92      if(_channel!=null){
93      try{
94      _channel.close();
95      }catch(IOException e){}
96      }
97      }
98      return result;
99      }
100     private SocketChannel connect()throws IOException{
101     InetSocketAddress socketAddress=
102     new InetSocketAddress("localhost",10000);
103     return SocketChannel.open(socketAddress);
104     }
105
```

```
106     private void sendSumRequest(int first,int second)throws
107 IOException{
108     _buffer.clear();
109     _intBuffer.put(0,first);
110     _intBuffer.put(1,second);
111     _channel.write(_buffer);
112     System.out.println("发送加法请求 "+first+"+"+second);
113     }
114
115     private int receiveResponse()throws IOException{
116     _buffer.clear();
117     _channel.read(_buffer);
118     return _intBuffer.get(0);
119     }
120     public static void main(String [] args){
121     SumClient sumClient=new SumClient();
        System.out.println("加法结果
    为 :"+sumClient.getSum(100,324));
        }
        } // SumClient
```

3. 非阻塞的加法服务器

```
1     首先在 openChannel 方法中加入语句
2     _serverChannel.configureBlocking(false);//设置成为非
3 阻塞模式
4     重写 WaitForConnection 方法的代码如下，使用非阻塞方式
5     private void waitForConnection()throws IOException{
6     Selector acceptSelector =
7 SelectorProvider.provider().openSelector();
8     /*在服务器套接字上注册 selector 并设置为接受 accept 方法
```

```
9  的通知。
10      这就告诉 Selector，套接字想要在 accept 操作发生时被放在
11 ready 表
        上，因此，允许多元非阻塞 I/O 发生。*/
        SelectionKey acceptKey = ssc.register(acceptSelector,
        SelectionKey.OP_ACCEPT);
        int keysAdded = 0;
```

原文链接：http://blog.sae.sina.com.cn/archives/2398?utm_source=tuicool

我为什么期待 M#？

评论很多，看来很多人误解了，希望大家能多去百度、bing、Google 一下在进行评论，也是对自己的负责。

首先本文的 M#跟这边所指的 m#不是一个东西 http://www.msharp.co.uk/

原因1：**M#还在研发中并没有正式发布。**

原因2：**网页中的 m#重在支持 asp.net，而微软定义中的 M#是一门编程语言当然不能局限于 asp.net**

原因3：**微软出的编程语言何时收费过？**

M#到底会不会脱离.net framework

M#到底会不会脱离.net framework 我也不清楚，只是猜测，一种美好的希望，如果连想都不能想是不是有点对不起社会？一个人没有了希望没有了理想还是一个人吗？

M#脱离.net framework 是很有可能的一件事

原因1：**M#是 Midori 系统的编程语言**

M#已经脱离了 Windows，至于有没有脱离 NT 内核暂时不做猜想，连 Windows 都脱离了为什么不能脱离.net framework？

摘抄：**同时微软目前也正在尝试寻找 Midori 与 Windows 系统之间的兼容性，让 Midori 的应用程序与 Windows 程序实现共存和互操作，并提供程序移植的方法。**

原因2：**M#将获得更好的性能**

除了小白大家都知道 C#、VB.NET、F#、J#都会被编译成 IL 丢给 CLR 去执行，如果 M#能提升性能并且没有脱离.NET Framework 那么 C#、VB.NET 等其他语言也一样能享受到这个待遇，既然如此为什么 M#研发团队会说将比 C#获得更好的性能？

原因3：**M#从2008年开始研究**

如果只是基于.net framework 那么需要5年的时间来研究语法吗？如果是这

样是不是太没有效率了？

它是 C#的补充

报导里面这么说：**“它在 C#的基础上添加了系统编程特性，可用来构建各种类型的应用，尤其是云计算应用。”**

用 C#开发了三年，总觉得缺少了些什么，虽然 C/S、B/S 都能做但还是觉得少了些什么，期间有考虑过是否尝试 c/c++、虽然 C#的语法我非常喜欢，是我接触到编程语言中最喜欢的一个，特别是 Linq。减少了大量的代码。

那究竟是少了点什么呢？

x1：x2你不是做开发的么？帮我写一个木马我要能看到我女朋友的桌面。

x2：这个简单，明天给你程序。

……第二天……

x2：x1我写好了，你拿去你女朋友机器上直接运行就好了，但是操作系统必须是 vista 以上的，然后需要安装.net framework 4.5。

x1：怎么看系统是不是 vista？如果不是 vista 以上怎么办？.net framework 4.5是个啥？我怎么安装.net framework 4.5失败呢？你写的程序真糟糕。

x2：……

.net framework，我觉得.net framework 是个负担，其实更多时候我觉得是.net framework 拖累的 C#，为了兼顾 VB.NET、F#、C#等语言.net framework 实在是太臃肿了，如果 C#增加了新特性那么.net framework 就需要同时为 VB.NET、F#等语言添加对应的实现，虽然中间有 IL，但还是太臃肿了，需要同时改变多个编译器，导致各个语言发现了发展瓶颈，维护量大增的问题。

总而言之：C#（.net framework）能做的事情太少了。

更好的性能

“Joe Duffy 表示，M#相对 C#、Java 等其他语言来说，它能在“性能”、“安全与生产力”两方面会达到更好的平衡。”

这其实也牵扯到.net framework 的问题，但这里部分地方不明示，大家心里明白就好了。

c#虽然拥有较好的性能但还是不够，至少对于目前来说，虽然现有的 CLR

可以根据即时环境（详情请看：“在.net 中为什么第一次执行会慢？”）动态生成最优的本地码，但对目前 CLR 对此的处理能力还是太弱了，导致了性能不够理想，而这一点 M#团队已经注意到了，他们会寻找到一个平衡点（**我琢磨着他们会把 M#编译成本地码,而可以同时兼容现有的.NET 类库,只不过引用现有的.NET 类库需要安装.net framework**）。

开源

“同时，他也表示 M#最终会开源，有可能就在令人充满期待的2014年”

如果想把 C#开发的程序迁移至 Liunx、OSX 上在以前几乎是不可能的事情，但现在虽然有了 Mono 可以做到还是会出现很多问题，这一点对于 M#来说就无需担心，因为它是开源的这边虽然不能预测 M#的开源级别与程度，但可以知道的是至少比现在容易，到那时候我们就可以少了一项从 Windows 转向 Liunx、OSX 的障碍了（**不知道为什么最近想从 Windows 转向 Liunx Or OSX**）。

M#已经研究了数年

据报道，微软研究 M#以长达4~5年（**多个报导时间不一样，期待有人能给出正确答案**），为其数年的编程语言值得期待。

它是微软的

微软虽然有很多的骂名，但不得不说他为开发者所做的，虽然现在的生态环境还远不如 java，但他为开发人员带来的便利是极大的，Visual Studio、IIS、SQL Server 等都具有良好的可视化界面及辅助工具，这些都能极大的减少开发者的时间。

写在最后

我不推崇任何技术，我只按需所取，我不黑微软也不捧微软，我只是说出我的想法，我想要的编码环境。

原文链接：http://www.cnblogs.com/ants/p/3508382.html?utm_source=tuicool

为什么大神级程序员的 C 语言代码里到处都是 goto?

当我学 C 语言时，老师整天告诉我："不要使用 goto，这是一个坏习惯，这种写法很烂，而且很危险!"等等。

但是为什么那么多内核程序员那么喜欢用 goto 呢？在这段 linux 内核 https://github.com/torvalds/linux/blob/master/kernel/sched/clock.c 代码里，我觉得可以用简单的一个 while 替换掉，如：

```
    while(condition) {

}

//或

do {

} while(condition);
```

注*这段代码来自 torvalds 的 linux 内核代码，其实不仅可以使用 while，还有很多地方可以使用 if () {} else {} 的结构代替，很多内核的其他文件也是如此，如 fs.open http://lxr.linux.no/linux+v3.12.6/fs/open.c#L464 对此我很不理解,在某些情况下使用 goto 比 while/do-while 好吗？如果是的话，为什么呢？

注： 这也许从另一个角度诠释了差点的更好 Worse Is Better 的观点。

来自国外网友的精彩评论：

回答一：

对于这个例子中，我估计是从原来 SMP 不安全(non-SMP-safe)的方式改成 SMP 的方式。使用 goto 语句对原来的代码改动量最小，引起潜在风险的概率最小。我其实也不赞成你们用这种方式，但我认为绝对不要使用 goto 也是一种误导。在一个只会向前走，绝不会后退的函数里，使用 goto 绝对不会引起死循环，

而且这种方式绝对是最简单最清楚的跳转方式。（如通过在清理代码和返回错误时使用）

回答二：

历史：我们也许记得 Dijkstra 在 1968 年写的 Goto Considered Harmfulhttp://www.u.arizona.edu/~rubinson/copyright_violations/Go_To_Considered_Harmful.html， 现在快半个世纪过去了。外面已经很少看到 goto 了。

不过我们来分析一下这个例子，一个关于错误处理的，让我们用结构化的语法来写：

```
    if (do_something() != ERR) {
    if (do_something2() != ERR) {
        if (do_something3() != ERR) {
            if (do_something4() != ERR) {
                ...
```

那么，换成 goto 呢？

```
    if (do_something() == ERR)  // 一行
    goto error;                 // |
if (do_something2() == ERR) // |
    goto error;                 // |
if (do_something3() == ERR) // |
    goto error;                 // V
if (do_something4() == ERR)   // 使用普通的平铺形式
    goto error;
```

我们看到这段代码都是平级的，不相互依辣的，明显 goto 的结构更好。

原文链接：

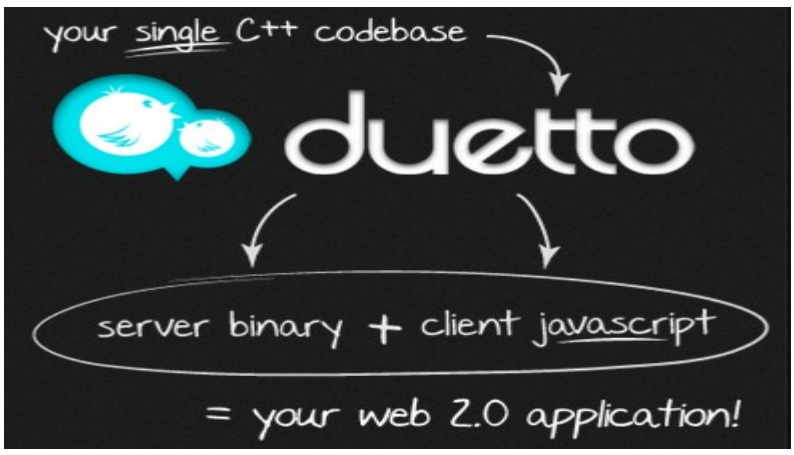
http://ourjs.com/detail/52ce07eb2caee88b29000002?utm_source=tuicool

阅读 Google 的 C++代码规范有感

函数参数顺序

C/C++函数参数分为输入参数和输出参数两种，有时输入参数也会输出（注：值被修改时）。输入参数一般传值或常数引用（const references），输出参数戒输入/输出参数为非常数指针（non-const pointers）。对参数排序时，将所有输入参数置于输出参数之前。不要仅仅因为是新添加的参数，就将其置于最后，而应该依然置于输出参数之前。这一点并不是必须遵循的规则，输入/输出两用参数（通常是类/结构体变量）混在其中，会使得规则难以遵守。

个人感受：这条规则相当重要，自己写代码的时候可能没有太大感觉，但是在阅读别人代码的时候感觉特别明显。如果代码按照这种规范来写，从某种角度来说，这段代码具有“自注释”的功能，那么在看代码的时候就会比较轻松。Doom3 的代码规范中提到，“Use ‘const’ as much as possible”，也是同样的意义。当然，const 除了阅读方便以外，还有个很重要的就是防止编码错误，一旦在程序中修改 const 变量，编译器就会报错，这样就减少了人工出错了可能性，这点尤为重要！



包含文件的名称及次序

将包含次序标准化可增强可读性、避免隐藏依赖（hidden dependencies，注：隐藏依赖主要是指包含的文件编译），次序如下：C 库、C++库、其他库的.h、项目内的.h。

项目内头文件应挑照项目源代码目录树结构排列，并且避免使用 UNIX 文件路径.（当前目录）和..（父目录）。

举例来说，google-awesome-project/src/foo/internal/fooserver.cc 的包含次序如下：

```
#include "foo/public/fooserver.h" // 优先位置
#include <sys/types.h>
#include <unistd.h>
#include <hash_map>
#include <vector>
#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/public/bar.h"
```

注意，对应的头文件一定要先包含，这样避免隐藏依赖，隐藏依赖的问题不懂的可以去 Google，网上有很多资料。另外，《C++编程思想》中提到的包含次序正好相反，从特殊到一般，但是有一点和 Google 代码规范是一样的，那就是对应的头文件是第一个包含。对于隐藏依赖的问题，以前只是习惯性的把对应的头文件放第一个，没有想过为什么，现在学习了……

作用域

全局变量

class 类型的全局变量是被禁止的，内建类型的全局变量是允许的，当然多线程代码中非常数全局变量也是被禁止的。永远不要使用函数返回值初始化全局变量。

不幸的是，全局变量的构造函数、析构函数以及初始化操作的调用顺序只是被部分规定，每次生成有可能会有变化，从而导致难以发现 bug。因此，禁止使用 class 类型的全局变量（包括 STL 的 string,vector 等），因为它们的初始化

顺序可能会导致出现问题。内建类型和由内建类型构成的没有构造函数的结构体可以使用，如果你一定要使用 class 类型的全局变量，请使用单件模式。

C++类

构造函数的职责

构造函数中只进行那些没有实际意义的初始化，可能的话，使用 Init()方法集中初始化为有意义(non-trivial)的数据。

个人感受：这种做法可以从一开始就避免一些 bug 的出现，或更容易解决一些 bug。构造函数+Init()函数初始化的方式与只用构造函数的方法相比，对计算机来说他们是没有区别的，但是人是会犯错的，这一条代码规范在某种程度上避免了一些人为错误，这个在开发中特别重要。

拷贝构造函数

仅在代码中需要拷贝一个类的对象的时候使用拷贝构造函数，不需要拷贝时使用 DISALLOW_COPY_AND_ASSIGN 这个宏(关于这个宏的内容，可以在网上搜到，我这里就不写了)。C++中对象的隐式拷贝是导致很多性能问题和 bugs 的根源。拷贝构造函数降低了代码可读性，相比按引用传递，跟踪按值传递的对象更加困难，对象修改的地方变得难以捉摸。

个人感受：和上一项的目的类似，为了避免人为错误！拷贝构造函数本来是为了方便程序员编程了，但是却有可能成为一个坑，为了避免这类问题，不需要拷贝时使用 DISALLOW_COPY_AND_ASSIGN，这样在需要调用拷贝构造函数的时候就会报错，减少了人为出错的可能性。C#和 Java 在这方面就做得比较好，虽然性能上不如 C++，但是人为出错的概率减少了很多。当然，使用一定的代码规范，可以在一定程度上减少 C++的坑。

继承

虽然 C++的继承很好用，但是在实际开发中，尽量多用组合少用继承，不懂的去看 GoF 的《Design Patterns》。

但重定义派生的虚函数时，在派生类中明确声明其为 virtual。这一条是为了为了阅读方便，虽然从语法的角度来说，在基类中声明了 virtual，子类可以不用再声明该函数为 virtual，但这样一来阅读代码的人需要检索类的所有祖先以确定该函数是否为虚函数 o(╯□╰)o。

多重继承

虽然允许，但是只能一个基类有实现，其他基类是接口，这样一来和 JAVA 一样了。这些东西在 C#和 JAVA 中都进行了改进，直接从语法上解决问题。C++的灵活性过高，也是个麻烦的问题，只能通过代码规范填坑。

接口

虚基类必须以 Interface 为后缀，方便阅读。阅读方便。

重载操作符

除少数特定情况外，不要重载操作符!!!“==”和“=”的操作 Euqals 和 CopyFrom 函数代替，这样更直观，也不容易出错。

个人感受：看到这一条，我有点惊讶，在学习 C++的时候，说重载操作符有神马神马好处，为什么现在又说不要重载操作符呢？仔细看了他的文档，确实说的有道理，导致可能出现的 bug 见其具体文档。在实际应用中，由于 C++的坑实在太多了，不得不把这种“好用”的东西干掉，因为出了 bug 又找不到，是一件很 0 疼的事情。

声明次序

1）typedefs 和 enums；

2）常量；

3）构造函数；

4）析构函数；

5）成员函数，含静态成员函数；

6）数据成员，含静态数据成员。

宏 DISALLOW_COPY_AND_ASSIGN 置于 private:块之后，作为类的最后部分。

其他 C++特性

引用参数

函数形参表中，所有的引用必须的 const！

个人感受：这么做是为了防止引用引起的误解，因为引用在语法上是值，却有指针的意义。虽然引用比较好用，但是牺牲其某些方面的特性，换来软件管理方面的便利，还是很值得了。

缺省参数

禁止使用函数缺省参数！

个人感受：看到这一点的时候觉得有点因噎废食了，其实缺省参数感觉还是蛮好用的。当然从另外一个角度来说，要使用 C++就不要怕这种小麻烦，如果因为使用这些特性造成了找不到的 bug，那会损失更多时间。

异常

不要使用 C++异常。

这一点我没有看懂，也许是因为它的异常机制没有 C#和 Java 那么完善吧……毕竟在 C#和 Java 里面异常还是很好用的东东。

除了记录日志，不要使用流，使用 printf 之类的代替。

这一条其实是有一些争议的，当然大多数人认为代码一致性比较重要，所以选择 printf，具体的可以看原文文档。

const 的使用

在任何可以的情况下都要使用 const。

这条规则赞一个,Doom3的代码规范里也提到了这一条。这么做有两个好处，一个是防止程序出错，因为修改了 const 类型的变量会报错；另一个就是方便阅读，使代码“自注释”。虽然这么做也有坏处，当然，总体来说利大于弊。

命名约定

1、总体规则：不要随意缩写，如果说 ChangeLocalValue 写作 ChgLocVal 还有情可原的话，把 ModifyPlayerName 写作 MdfPlyNm 就太过分了，除函数名可适当为动词外，其他命名尽量使用清晰易懂的名词；

2、宏、枚举等使用全部大写+下划线；

3、变量（含类、结构体成员变量）、文件、命名空间、存取函数等使用全部小写+下划线，类成员变量以下划线结尾，全局变量以 g_开头；

4、普通函数、类型（含类与结构体、枚举类型）、常量等使用大小写混合，不含下划线；

使用这套命名约定，可以使代码具有一定程度的“自注释”功能，方便他人阅读，也方便自己以后修改。当然3、4两点也可以使用其他的命名约定，只要团队统一即可。

格式

1、行宽原则上不超过80列，把22寸的显示屏都占完，怎么也说不过去；

2、尽量不使用非 ASCII 字符，如果使用的话，参考 UTF-8 格式（尤其是 UNIX/Linux 下，Windows 下可以考虑宽字符），尽量不将字符串常量耦合到代码中，比如独立出资源文件，返不仅仅是风格问题了；

3、UNIX/Linux 下无条件使用空格，MSVC 的话使用 Tab 也无可厚非；（我没用过 Linux，不懂为什么在 Linux 下无条件使用空格）

4、函数参数、逻辑条件、初始化列表：要么所有参数和函数名放在同一行，要么所有参数并排分行；

5、除函数定义的左大括号可以置于行首外，包括函数/类/结极体/枚举声明、各种语句的左大括号置于行尾，所有右大括号独立成行；

6、./->操作符前后丌留空格，*/&不要前后都留，一个就可，靠左靠右依各人喜好；

7、预处理指令/命名空间不使用额外缩进，类/结构体/枚举/函数/语句使用缩进；

8、初始化用=还是()依个人喜好，统一就好；

9、return 不要加()；

10、水平/垂直留白不要滥用，怎么易读怎么来。

写在最后

总的来说，这套代码规范还是相当不错的，既有防止错误使用 C++的某些特性而导致 bugs 的规范，又有代码书写的相关规范使其便于阅读，建议搞 C++的童鞋都看一看。当然，具体的团队应该会有具体的代码规范，代码风格方面大家可能会有一些区别；不使用 C++某些特性（比如不使用 C++异常，禁止使用函数缺省参数)方面，应该按照具体情况进行折中处理，而不应该生搬硬套代码规范；但是“不将字符串常量耦合到代码中”这种规范，是大家必须遵守的。

原文链接：

http://www.cnblogs.com/wangchengfeng/p/3503820.html?utm_source=tuicool

[程序设计]

iOS- CoreData 数据库管理利器！

1. 前文

上次用 SQLite3实现了数据管理，这次准备用 CoreData 来实现。

Core Data 是 iOS SDK 里的一个很强大的框架，允许程序员以面向对象的方式储存和管理数据。使用 Core Data 框架，程序员可以很轻松有效地通过面向对象的接口管理数据

相比 SQLite3来说，用 CoreData 更有利于程序员来管理数据，

除了开头的准备工作略微繁琐点，后面的操作都很方便。

而且在 CoreData 在数据操作过程中,无需编写任何 SQL 语句,这一点和 JAVA 里的 hibernate 框架类似。

那么，下面我就直接说说它的实现步骤。

2. CoreData 实现的主要步骤

2.1. 要使用 Core Data，首先需要导入 CoreData 框架

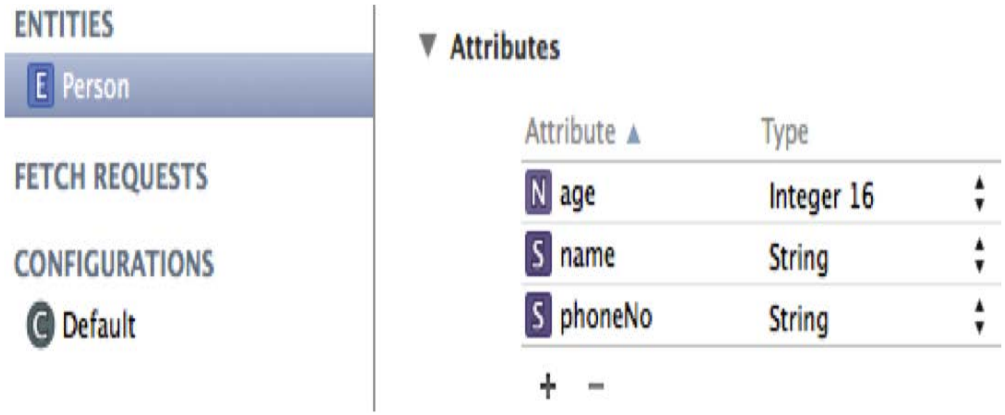
表结构：NSEntityDescription

表记录：NSManagedObject

数据库存放方式：NSPersistentStoreCoordinator(持久化存储协调者)

数据库操作：NSManagedObjectContext(被管理的对象上下文)

2.2. 接着要使用 Code Data，首先需要定义模型文件，描述应用程序中的所有实体（Entities）



2.3.创建连接数据库

- 首先需要创建一个操作数据库的上下文。NSManagedObjectContext
- 操作数据库的上下文需要设置一个调度者属性，这个调度者是用来将图形化建立的模型和数据库联系起来。
- 给调度者添加一个需要联系的数据库。

```
1 // Merging 合并可以将图形化建立的所有 Model 汇总到一个数据库文件中
2          NSManagedObjectModel  *model  =  [NSManagedObjectModel mergedModelFromBundles:nil];
3
4     // 调度者的实例化，需要 Model
5                   NSPersistentStoreCoordinator    *store    =  [[NSPersistentStoreCoordinator                                    alloc] initWithManagedObjectModel:model];
6
7     // 数据库是一个文件，持久化连接的文件
8     NSError *error = nil;
9     NSURL *url = [@"my.db" appendDocumentDirURL];
10
11    // 添加持久化存储的数据库
12          [store  addPersistentStoreWithType:NSSQLiteStoreType configuration:nil URL:url options:nil error:&error];
13
14    if (error == nil) {
15        NSLog(@"数据库建立成功");
16
17        // 获取到数据库操作的上下文，类似于 SQLite 的句柄
18        _sharedContext = [[NSManagedObjectContext alloc] init];
```

```
19
20          // 让上下文记录住存储调度
21          _sharedContext.persistentStoreCoordinator = store;
22      } else {
23          NSLog(@"数据库建立失败");
24      }
```

2.4.添加，更新，删除

添加：

1.新建实体 INST （插入）

```
1    Person *p = [[Person alloc] init]
```

2.设置实体的属性

```
1 // 设置对象内容
2      person.name = _nameText.text;
3      person.phoneNo = _phoneText.text;
4      person.qq = _qqText.text;
5      person.weibo = _weiboText.text;
```

3.保存上下文

```
1       //实体描述
2                                              [NSEntityDescription
insertNewObjectForEntityForName:@"Person"
inManagedObjectContext:context]
3
4      // 获取上下文
5            NSManagedObjectContext  *context  =  [[DataManager
sharedDataManager] sharedContext];
6
7      // 让上下文保存
```

```
    if ([context save:nil]) {
        NSLog(@"保存成功");

        // 返回上级视图控制器
                                        [self.navigationController
popViewControllerAnimated:YES];
    } else {
        NSLog(@"保存失败!");
    }
```

更新：

1.判断是否已有一模一样的模型

```
    //判断
  Person *person = _editPerson;


    // 如果 person == nil 表示是新建用户
    if (person == nil) {
        person                =                [NSEntityDescription
insertNewObjectForEntityForName:@"Person"
inManagedObjectContext:context];
    }
```

2.设置实体属性

```
   person.name = _nameText.text;
    person.phoneNo = _phoneText.text;
    person.qq = _qqText.text;
    person.weibo = _weiboText.text;
```

3.保存上下文

```
    // 让上下文保存
```

```
    if ([context save:nil]) {
        NSLog(@"保存成功");

        // 返回上级视图控制器
                                        [self.navigationController popViewControllerAnimated:YES];
    } else {
        NSLog(@"保存失败!");
    }
```

删除

```
// 1. 首先找到要删除哪条记录
        Person *person =
// 2. 删除
             NSManagedObjectContext *context = [[DataManager sharedDataManager] sharedContext];

        // 让上下文删除
        [context deleteObject:person];

        // 上下文保存
        if ([context save:nil]) {
            NSLog(@"删除成功!");
        } else {
            NSLog(@"删除失败!");
        }
```

2.4. 查询

三. 查询

1. 使用 NSFetchedResultsController 控制器

```
// 查询结果控制器
    NSFetchedResultsController *_fetchedResultsController;
```

2. 监控 managed object context 对象的改变，报告给 delegate

```
1  // 设置代理
2      _fetchedResultsController.delegate = self;
```

2.1当操作数据上下文的内容改变的时候，会自动调用抓取结果控制器的代理方法

```
1 #pragma mark 查询结果控制器代理方法
2 - (void)controllerDidChangeContent:(NSFetchedResultsController *)controller
3 {
4     // 新增、修改、删除
5     [self.tableView reloadData];
6 }
```

3. 创建控制器

一般来说,你会创建一个 NSFetchedResultsController 实例作为 tableview 的成员变量。初始化的时候，你提供四个参数：

1。一个 fetchrequest. 必须包含一个 sortdescriptor 用来给结果集排序。

2。 一个 managedobject context。 控制器用这个 context 来执行取数据的请求。

3。 一个可选的 keypath 作为 sectionname。控制器用 keypath 来把结果集拆分成各个 section。(传 nil 代表只有一个 section)

4。 一个 cachefile 的名字，用来缓冲数据，生成 section 和索引信息。

```
1                    NSFetchRequest          *reqest          = _fetchedResultsController.fetchRequest;
1       1> 查询请求
2       2> 数据库上下文
3       3> 表格中用于分组的字段名
4       4> 缓存名称
```

```
5      */
6       _fetchedResultsController = [[NSFetchedResultsController
alloc]    initWithFetchRequest:request    managedObjectContext:context
sectionNameKeyPath:nil cacheName:nil];
```

5.注意：一定要执行抓取请求，返回的数据在 sections 里，这个数组中装的都是遵守 NSFetchedResultsSectionInfo 这个协议的对象。通过 numberOfObjects 就能获取一组有多少数据对象了。

```
1              return      [_fetchedResultsController.sections[0]
numberOfObjects];
```

原文链接：http://www.cnblogs.com/qingche/p/3512859.html?utm_source=tuicool

程序媛也话Android 之 自定义控件(垂直方向滑动条)

Android 里已经有足够多的控件供开发者使用，但有时候我们还是会想要一些不一样的东西，比如一些 UI 特效，比如一些3D 动画，今天就讲讲比较 basic 的东西：自定义控件。

1.效果图

如果项目里需要一个通用的控件，然后 UI 给你这样一个效果图，你接下来会打算怎么做？

用户可以按住拖动

点击要切换的状态，然后自动滑动到那一端

（本来是没有这个效果图的，又不想一张张贴不同的状态，就画了一下这个 gif 图，关于怎么在 ubuntu 下画 gif 图，可以看一下下面这篇）

程序媛也会画图 之 在 ubuntu 下用 GIMP 制作 gif

2. 分析

看一下有没有现成的 widget，这似乎和 android.widget.Switch 有点类似，可是 Swithc 是水平的，水平没有关系，改成垂直的问题不大，先来尝试下好了，就先把背景和 button 的图片换一下，来看一下结果是怎样：

额。。。这个切换似乎生硬了点，没有渐变的动画。好吧，那还是重新自己写一个控件吧。

3. 创建 Andriod 自定义控件的步骤

怎么建立一个自定义的控件，说起来并不难，有三个内容需要实现：

3.1新建一个控件类，继承 android.view.View 类：

```
1 public class XXXView extends View {
2     ...
3     protected void onDraw(Canvas canvas) {
4         ...
5     }
6
7     public boolean onTouchEvent(MotionEvent event) {
8         ...
```

```
    }

    public interface OnXXXListener { //状态回调，同View.OnClickListener
        public abstract void xxx();
        public abstract void xxx();
    }
}
```

3.2 在布局文件 xml 里使用这个控件：

```
<com.xxx.xxx.XXXView  android:id="@+id/xxx"
    android:layoutWidth="..."
    android:layoutHeight="..." >
</com.xxx.xxx.XXXView>
```

3.3 在 Activity 类里获得这个控件：

```
mXXXView = (XXXView) findViewById(R.id.xxx);
mXXXView.setListener(mXXXViewListener);
```

以上这简单的3个步骤就是创建和使用控件的内容了，到这里，如果你是个喜欢着急写代码的人，你也可以先搭一个程序框架出来跑跑看啦。

4.考虑怎么画？

4.1拖动

用户需要能拖动 Button，那也就是说我们在控件里需要捕获用户的 touch event，知道用户到底是做了什么动作(ACTION_DOWN, ACTION_MOVE, UP)，还有操作的位置在哪里（getX(), getY()）.

这些信息从哪里可以知道？--》onTouchEvent()回调！

4.2动画

动画的本质就是图片+位置+时间差。

在效果图中，用户也可以点击一个状态，让控件滑动。那这个滑动的过程就是一个动画的。

图片我们有，那怎么把图片画到 Canvas 上？-》在 onDraw()回调里面画。在主线程里只要调用 invalidate()，就会重新触发 onDraw()的执行。如果我们在一定的时间间隔，在不同的位置重新画图片，不就是动画了？

位置可以从用户行为获得，或者自己计算；

时间差，在 Android 里面控制时间最容易的是什么？当然是 Handler 啦，因为它可以发送 delay 的消息。

4.3渐变的实现

效果图中还有个渐变的过程，这个看起来好像蛮麻烦，其实也好办。因为有 Alpha 的存在。我们可以在画的时候根据不同的位置，设置 Paint 不同的 Alpha 值，一个图片 Alpha 慢慢减小，另一个图片 Alpha 慢慢增大。ok，分析到这里，就大概知道该怎么做了，在 onTouchEvent()回调里，获得用户的行为和位置，并记录下来，在适当的时候发送 Message 给 Handler，或者直接调用 invalidate()重新画。在 Handler 里，接收到信息，就根据当前的状态，更新图片下一个应该出现的位置，然后调用 invalidate()触发重新画。

5.计算位置

ok，上面已经确定以什么方式做了，接下来就要用到一点点数学的计算了。我们要确定图片从哪里开始动，动到哪里结束，还有在什么位置开始切换状态。先切下图：


（文字也做成了图片，其实凡是涉及到文字的都不应该做成图片，如果有人切换到中文，然后他又不认识 on off 呢，而且这些文字应该要可设置的才对。这里图方便就做成图片了。）

然后就是一些重要坐标位置啦：

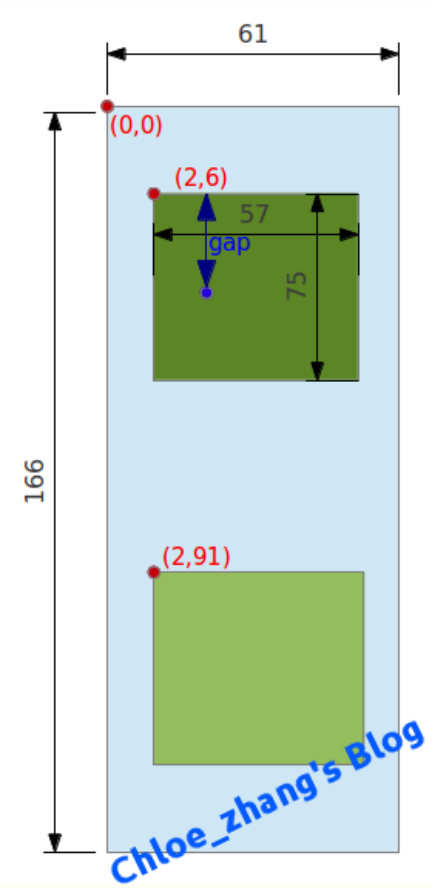


图1　蓝色是那个长条的图片，绿色两块是在两个状态下 Button 所在的位置。

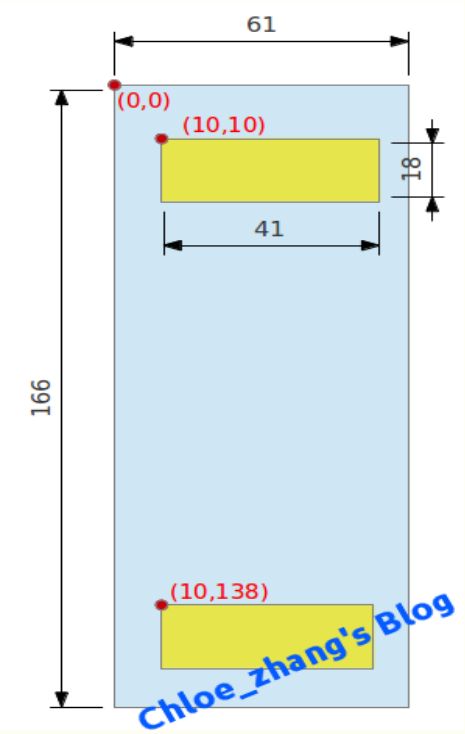


图2　黄色的区域是两个小的灰色文字图片

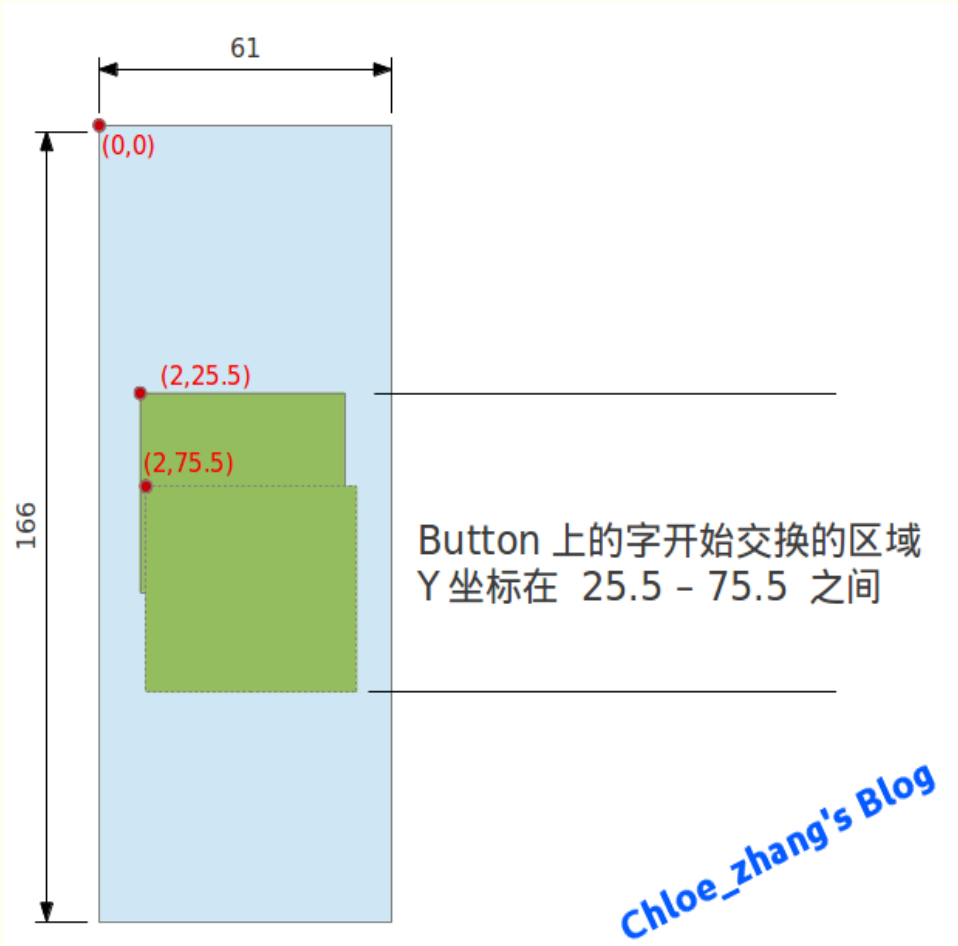


图3　这个区域就是文字开始切换的区域

6. 伪代码

现在方法也有了，数据也有了，就可以开始写代码了。

为了叙述方便，就用伪代码代替了，下面是最重要的三个部分的伪码：

处理用户行为的逻辑：

```
1 public boolean onTouchEvent(MotionEvent event) {  //处理用户行为
 2     case ACTION_DOWN:
 3         if (坐标在图1中蓝色区域) { //touch 在无效的区域
 4             return;
 5         }
 6
 7         if (坐标在图1中绿色区域中 Button 在的区域) { //当前状态是 on，就是上面的区域，否则，就是下面的区域
```

```
8               获得坐标与上边缘的距离 gap;
9           } else {
10              设置正在滑动标志;
11               设置动画的方向，发送 Message;    //会执行到这里的情况是，比如当前状态是 on，用户点击了 off 那一端，那接下来控件就要自动滑动切换到 off 状态。
12          }
13          break;
14
15      case ACTION_MOVE:
16          if (上次 Down 是在无效区域 | 正在切换状态) { //此时不用响应 Move 动作。
17              return;
18          }
19
20          if (根据当前的坐标计算，滑块将不在背景区域) {
21              return;
22          }
23
24          if (根据当前的坐标计算，在文字交换的区域) {
25              设置交换标记;
26          }
27          记录滑块当前位置;
28          invalidate();
29          break;
30
31      case ACTION_UP:
32          if (上次 Down 是在无效区域 | 正在切换状态) { //此时不用响应 Up 动作。
```

```
33                 return;
34             }
35
36             取消交换标识;
37              if(根据当前坐标计算，最后的状态是 on) {
38                 设置滑块位置为 on 状态时的位置;
39                 修改状态为 on;
40                 invalidate();
41              } else {
42                 设置滑块位置为 off 状态时的位置;
43                 修改状态为 off;
44                 invalidate();
45             }
46 }
```

处理自动滑动：

```
1 private Handler mHandler = new Handler() {  //用于处理自动滑动那部分逻辑
2     public void handleMessage(Message msg) {
3          if (计数 > 20) {
4              设置当前状态;
5              设置滑块的位置;
6              取消正在滑动的标志;
7              计数归0;
8               return;
9          }
10
11          根据计数，获得 interpolator.getInterpolation; //这里用
```

了 AccelerateDecelerateInterpolator，让动画有一个加速的效果，其实这么短的距离效果看不出来。

```
            计算滑块的位置;
            invalidate();
             计数+1;
             sendMessageDelayed(0, 20); //20ms 后画下一帧。
        }

};
```

画：

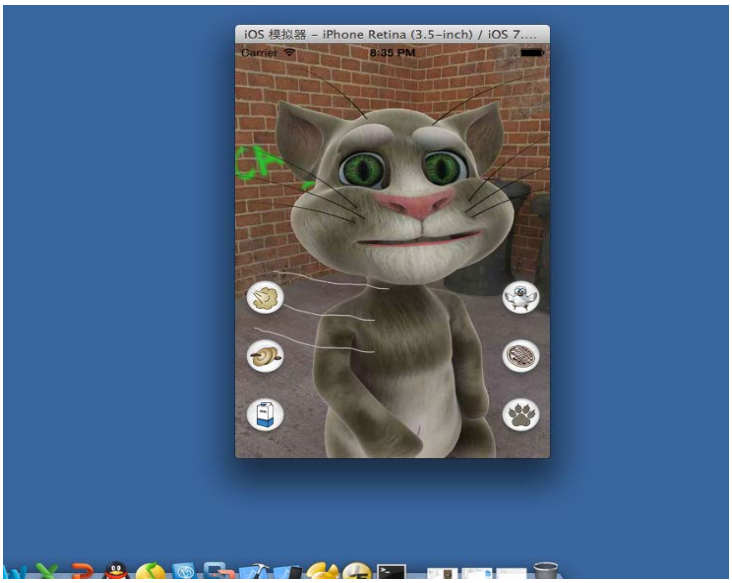
```
protected void onDraw(Canvas canvas) { //具体画的代码
    画背景;

     if (在状态交换区域) {
        根据滑块位置这是 Paint 的 Alpha 值;
         用上面设置的 Paint 画那四个小图; //在状态交换的时候，四个小图都是显示的。
     } else {
        根据当前的状态，画 on 滑块或 off 滑块;
    }
}
```

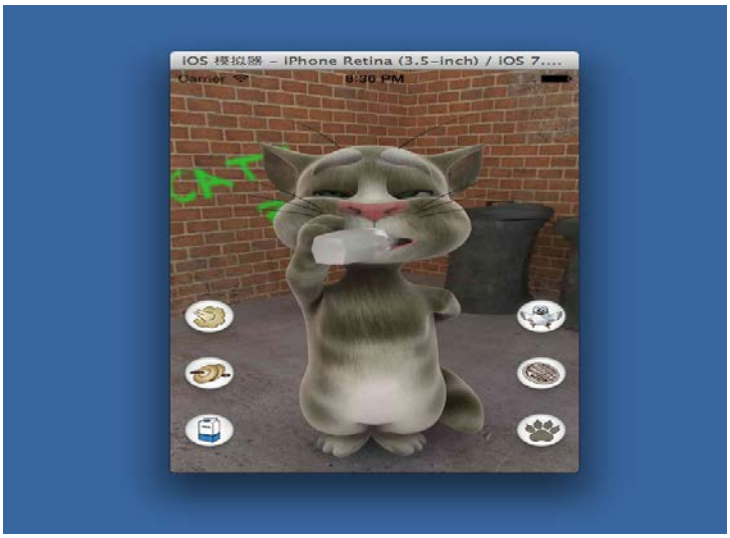
ok，有上面3部分的内容，基本上就可以了。

原文链接：

http://www.cnblogs.com/zhangxinyan/p/3513723.html?utm_source=tuicool

iOS- 利用 UIImageView 自己整了个不会说话的汤姆猫





1. 实现思路

先说说我实现它的主要思路，很简单，主要利用 UIImageView 连续动画播放，和按钮的点击事件，就可以完成了这么一个简单的不会说话的汤姆猫。

2. 实现细节

2.1. 加载本地字典里保存的本地图片名

```
@property (weak, nonatomic) IBOutletUIImageView *tom;
NSDictionary *_dict; // 保存所有图片的个数
```

```
1 // 1.获得 tom.plist 的全路径
2     NSBundle *bundle = [NSBundle mainBundle];
```

```
3           NSString *path = [bundle pathForResource:@"tom"
ofType:@"plist"];
4
5     // 2.根据文件路径加载字典
6     _dict = [NSDictionary dictionaryWithContentsOfFile:path];
```

2.2.抽取动画连续播放的方法出来

1.有缓存（无法释放,参数传的是文件名）

```
[UIImage imageNamed:@""];
```

2.无缓存（用完就会释放，参数传的是全路径）

```
 [[UIImage alloc] initWithContentsOfFile:];
```

```
    // 1.创建可变数组
    NSMutableArray *images = [NSMutableArray array];

    // 2.添加图片
    for (int i = 0; i<count; i++) {
        // 图片名
        NSString *name = [NSString stringWithFormat:@"%@_%02d.jpg",
filename, i];
        // 全路径
        NSString *path = [[NSBundle mainBundle]
pathForResource:name ofType:nil];

        // 加载图片(缓存)
//        UIImage *img = [UIImage imageNamed:name];
        // 没有缓存
        UIImage *img = [[UIImage alloc]
initWithContentsOfFile:path];
```

```
            [images addObject:img];
        }
        // 3.设置动画图片(有顺序)
        _tom.animationImages = images;// 序列帧动画
        // 4.只播放一次
        _tom.animationRepeatCount = 1;
        // 5.设置动画的持续时间
        _tom.animationDuration = 0.1 * count;
        // 5.开始动画
        [_tom startAnimating];
```

2.3.监听按钮的点击，实现图片的连续播放形成动画

```
1 #pragma mark 监听所有的按钮点击
2 - (IBAction)btnClick:(UIButton *)sender {
3     // 1.如果tom正在播放动画，直接返回
4     if (_tom.isAnimating) return;
5
6     // 2.取出按钮文字
7                         NSString     *title     =     [sender
titleForState:UIControlStateNormal];
8
9     // 3.获得图片数量
10    int count = [_dict[title] intValue];
11
12    // 4.播放动画
13   [self playAnim:count filename:title];
14 }
```

原文链接：http://www.cnblogs.com/qingche/p/3511313.html?utm_source=tuicool

Android 捕获全局异常信息并实现上传

在做项目时，经常会把错误利用异常抛出去，这样在开发时就可以通过手机抛出的异常排查错误。但是当程序开发完毕，版本稳定，需要上线时，为了避免抛出异常影响用户感受，可以用 UncaughtExceptionHandler 捕获全局异常，对异常做出处理。比如我们可以获取到抛出异常的时间、手机的硬件信息、错误的堆栈信息，然后将获取到的所有的信息发送到服务器中，也可以发送到指定的邮件中，以便及时修改 bug。

示例：

自定义异常类实现 UncaughtExceptionHandler 接口，当某个页面出现异常就会调用 uncaughtException 这个方法，我们可以在这个方法中获取异常信息、时间等，然后将获取到的信息发送到我们指定的服务器

Java 代码

```
/**
 * 自定义的 异常处理类 ， 实现了 UncaughtExceptionHandler 接口
 * @author Administrator
 *
 */
public class MyCrashHandler implements UncaughtExceptionHandler {
    // 需求是 整个应用程序 只有一个 MyCrash-Handler
    private static MyCrashHandler myCrashHandler ;
    private Context context;
    private DoubanService service;
    private SimpleDateFormat dataFormat = new SimpleDateFormat("yyyy-MM-dd-HH-mm-ss");

    //1.私有化构造方法
```

```
    private MyCrashHandler(){

    }

    public static synchronized MyCrashHandler getInstance(){
        if(myCrashHandler!=null){
            return myCrashHandler;
        }else {
            myCrashHandler  = new MyCrashHandler();
            return myCrashHandler;
        }
    }
    public void init(Context context,DoubanService service){
        this.context = context;
        this.service = service;
    }


    public void uncaughtException(Thread arg0, Throwable arg1)
{
        System.out.println("程序挂掉了 ");
        // 1.获取当前程序的版本号. 版本的 id
        String versioninfo = getVersionInfo();

        // 2.获取手机的硬件信息.
        String mobileInfo  = getMobileInfo();

        // 3.把错误的堆栈信息 获取出来
        String errorinfo = getErrorInfo(arg1);
```

```
            // 4.把所有的信息 还有信息对应的时间 提交到服务器
            try {
                service.createNote(new
PlainTextConstruct(dataFormat.format(new Date())),
                        new
PlainTextConstruct(versioninfo+mobileInfo+errorinfo), "public",
"yes");
            } catch (Exception e) {
                e.printStackTrace();
            }

            //干掉当前的程序

android.os.Process.killProcess(android.os.Process.myPid());
        }

        /**
         * 获取错误的信息
         * @param arg1
         * @return
         */
        private String getErrorInfo(Throwable arg1) {
            Writer writer = new StringWriter();
            PrintWriter pw = new PrintWriter(writer);
            arg1.printStackTrace(pw);
            pw.close();
            String error= writer.toString();
            return error;
```

```
    }

    /**
     * 获取手机的硬件信息
     * @return
     */
    private String getMobileInfo() {
        StringBuffer sb = new StringBuffer();
        //通过反射获取系统的硬件信息
        try {

            Field[] fields =
Build.class.getDeclaredFields();
            for(Field field: fields){
                //暴力反射 ，获取私有的信息
                field.setAccessible(true);
                String name = field.getName();
                String value = field.get(null).toString();
                sb.append(name+"="+value);
                sb.append("\n");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return sb.toString();
    }

    /**
     * 获取手机的版本信息
```

```
     * @return
     */
    private String getVersionInfo(){
        try {
            PackageManager pm = context.getPackageManager();
             PackageInfo info
=pm.getPackageInfo(context.getPackageName(), 0);
              return  info.versionName;
        } catch (Exception e) {
            e.printStackTrace();
            return "版本号未知";
        }
    }
}
```

创建一个 Application 实例将 MyCrashHandler 注册到整个应用程序上，创建出服务并进行传递：

Java 代码

```
/**
 * 整个(app)程序初始化之前被调用
 * @author Administrator
 *
 */
public class DoubanApplication extends Application {
    public NoteEntry entry;
    @Override
    public void onCreate() {
        super.onCreate();
        String apiKey = "0fab7f9aa21f39cd2f027ecfe65dad67";
        String secret = "87fc1c5e99bfa5b3";
```

```
        // 获取到 service
        DoubanService myService = new DoubanService("我的小豆豆", apiKey,
                secret);
        myService.setAccessToken("1fa4e5be0f808a0b5eeeb13a2e819e21",
"56a622c1138dbfce");
        MyCrashHandler handler = MyCrashHandler.getInstance();
        handler.init(getApplicationContext(),myService);
        Thread.setDefaultUncaughtExceptionHandler(handler);
    }
}
```

原文链接：http://hunankeda110.iteye.com/blog/2001286?utm_source=tuicool

iOS7 如何解决 iOS 瀑布流运行不流畅

如果 UITableView 滑动太快，可能同时就发出了比如10个图片请求。这些请求虽然都在后台运行，但是它们可能在同一个时间点返回 UI 线程。这个时候如果加载图片到 UIImageView 太频繁，就会造成 UI 卡得严重。（虽然在 new iPad 和 iPhone4s 上看不出来）

在找到这个问题的同时，也发现 performSelectorAfterDelay 这个方法，会堆积到 UI 线程空闲的时候执行。而 dispatch_after 或者 dispatch_async 都会直接插入 UI 线程当场执行。所以这个问题其实可以用 performSelectorAfterDelay 来解决，测试也是非常流畅，感觉不出一点点的卡。

但会出现新的问题，那就是在滑动过程中，不会加载任何图片。知道 scrollView 停止的时候，图片才会出来。当然这不是理想的解决方法了。这个方法也没有解决异步过程集中到达 UI 线程的问题。然后采用了 NSOperationQueue 来解决这个问题。

问题本身和 UITableView 加载不流畅是一样的。

解决办法

主要要做到一下几个方面：

除了 UI 部分，所有的加载操作都在后台完成。

这一点可以通过 dispatch 或者 performSelectorInBackground 或者 NSOperationQueue 来实现。见：

在 iOS 开发中利用 GCD 进行多线程编程

iOS 开发中使用 NSOperationQueue 进行多线程操作

避免后台加载完成多个资源之后集中到达占用 UI 线程的处理时间太长。这一点可以通过 NSOperationQueue 来实现，将资源到 UI 的展现过程放在队列中逐个执行，且在每个操作完成之后进行强制等待，可以用 usleep(int microSeconds)来解决。

重用 cell。

创建 cell 一般是很慢的，一定要重用，甚至为了 performance，可以在 view 创建之初就创建足够多的 cell 在重用队列中。

原文链接：http://www.gowhich.com/blog/527?utm_source=tuicool

[后端架构]

12 款免费与开源的 NoSQL 数据库介绍

Naresh Kumar 是位软件工程师与热情的博主，对于编程与新事物拥有极大的兴趣，非常乐于与其他开发者和程序员分享技术上的研究成果。近日，Naresh 撰文谈到了12款知名的免费、开源 NoSQL 数据库，并对这些数据库的特点进行了分析。

现在，NoSQL 数据库变得越来越流行，我在这里总结出了一些非常棒的、免费且开源的 NoSQL 数据库。在这些数据库中，MongoDB 独占鳌头，拥有相当大的使用量。这些免费且开源的 NoSQL 数据库具有很好的可伸缩性与灵活性，非常适合于大数据存储与处理。相较于传统的关系型数据库，这些 NoSQL 数据库在性能上具有很大的优势。然而，这些 NoSQL 数据库未必最适合你。大多数常见的应用仍然可以使用传统的关系型数据库进行开发。NoSQL 数据库依然不太适合于那些任务关键型的事务要求。我对这些数据库进行了一些简单介绍，下面就来看看。

1. MongoDB

MongoDB 是个面向文档的数据库，使用 JSON 风格的数据格式。它非常适合于网站的数据存储、内容管理与缓存应用，并且通过配置可以实现复制与高可用性功能。

MongoDB 具有很强的可伸缩性，性能表现优异。它使用 C++编写，基于文档存储。此外，MongoDB 还支持全文检索、跨 WAN 与 LAN 的高可用性、易于实现的复制、水平扩展、基于文档的丰富查询、在数据处理与聚合等方面具有很强的灵活性。

2. Cassandra

这是个 Apache 软件基金会的项目，Cassandra 是个分布式数据库，支持分散的数据存储，可以实现容错以及无单点故障等。换句话说，“Cassandra 非常适合于那些无法忍受数据丢失的应用”。

3. CouchDB

这也是 Apache 软件基金会的一个项目，CouchDB 是另一个面向文档的数据库，以 JSON 格式存储数据。它兼容于 ACID，像 MongoDB 一样，CouchDB 也可以

用于存储网站的数据与内容，以及提供缓存等。你可以通过 JavaScript 在 CouchDB 上运行 MapReduce 查询。此外，CouchDB 还提供了一个非常方便的基于 Web 的管理控制台。它非常适合于 Web 应用。

4. Hypertable

Hypertable 模仿的是 Google 的 BigTable 数据库系统。Hypertable 的创建者将“成为高可用、PB 规模的数据库开源标准”作为 Hypertable 的目标。换言之，Hypertable 的设计目标是跨越多个廉价的服务器可靠地存储大量数据。

5. Redis

这是个开源、高级的键值存储。由于在键中使用了 hash、set、string、sorted set 及 list，因此 Redis 也称作数据结构服务器。这个系统可以帮助你执行原子操作，比如说增加 hash 中的值、集合的交集运算、字符串拼接、差集与并集等。Redis 通过内存中的数据集实现了高性能。此外，该数据库还兼容于大多数编程语言。

6. Riak

Riak 是最为强大的分布式数据库之一，它提供了轻松且可预测的伸缩能力，向用户提供了快速测试、原型与应用部署能力，从而简化应用的开发过程。

7. Neo4j

Neo4j 是一款 NoSQL 图型数据库，具有非常高的性能。它拥有一个健壮且成熟的系统的所有特性，向程序员提供了灵活且面向对象的网络结构，可以让开发者充分享受到拥有完整事务特性的数据库的所有好处。相较于 RDBMS，Neo4j 还对某些应用提供了不少性能改进。

8. Hadoop HBase

HBase 是一款可伸缩、分布式的大数据存储。它可以用在数据的实时与随机访问的场景下。HBase 拥有模块化与线性的可伸缩性，并且能够保证读写的严格一致性。HBase 提供了一个 Java API，可以实现轻松的客户端访问；提供了可配置且自动化的表分区功能；还有 Bloom 过滤器以及 block 缓存等特性。

9. Couchbase

虽然 Couchbase 是 CouchDB 的派生，不过它已经成为了一款功能完善的数据库产品。它向文档数据库转移的趋势会让 MongoDB 感到压力。每个节点上它都是

多线程的，这是个非常主要的可伸缩性优势，特别是当托管在自定义或是Bare-Metal 硬件上时更是如此。借助于一些非常棒的集成特性，诸如与 Hadoop 的集成，Couchbase 对于数据存储来说是个非常不错的选择。

10. MemcacheDB

这是个分布式的键值存储系统，我们不应该将其与缓存解决方案搞混；相反，它是个持久化存储引擎，用于数据存储并以非常快速且可靠的方式检索数据。它遵循 memcache 协议。其存储后端用于 Berkeley DB 中，支持诸如复制与事务等特性。

11. REVENDB

RAVENDB 是第二代开源数据库，它面向文档存储并且无模式，这样就可以轻松将对象存储到其中了。它提供了非常灵活且快速的查询，通过对复制、多租与分片提供开箱即用的支持使得我们可以非常轻松地实现伸缩功能。它对 ACID 事务提供了完整的支持，同时又能保证数据的安全性。除了高性能之外，它还通过 bundle 提供了轻松的可扩展性。

12. Voldemort

这是个自动复制的分布式存储系统。它提供了自动化的数据分区功能，透明的服务器失败处理、可插拔的序列化功能、独立的节点、数据版本化以及跨越各种数据中心的数据分发功能。

各位 InfoQ 读者，不知在你的项目中曾经、现在或是未来使用了哪些 NoSQL 数据库。现今的 NoSQL 世界纷繁复杂，NoSQL 数据库也多如牛毛，而且有一些数据库提供了相似的特性，本文所列出的只是其中比较有代表性的12款 NoSQL 产品。你是否使用过他们呢？是否使用了本文没有介绍的产品呢？他们有哪些特性打动了你，让你决定使用他们呢？非常欢迎将你的经历与看法与我们一起分享。

原文链接：

http://www.infoq.com/cn/news/2014/01/12-free-and-open-source-nosql?utm_source=tuicool

NoSQL 与 RDBMS：何时使用，何时不使用

时至今日，互联网上有数以亿计的用户。大数据与云计算已经成为很多主要的互联网应用都在使用或是准备使用的技术，这是因为互联网用户每天都在不断增长，数据也变得越来越复杂，而且有很多非结构化的数据存在，这是很难通过传统的关系型数据库管理系统来处理的。NoSQL 技术则能比较好地解决这个问题，它主要用于非结构化的大数据与云计算上。从这个角度来看，NoSQL 是一种全新的数据库思维方式。

为何要使用 NoSQL 数据库？

1. NoSQL 具有灵活的数据模型，可以处理非结构化/半结构化的大数据

现在，我们可以通过 Facebook、D&B 等第三方轻松获得与访问数据，如个人用户信息、地理位置数据、社交图谱、用户产生的内容、机器日志数据以及传感器生成的数据等。对这些数据的使用正在快速改变着通信、购物、广告、娱乐以及关系管理的特质。没有使用这些数据的应用很快就会被用户所遗忘。开发者希望使用非常灵活的数据库，能够轻松容纳新的数据类型，并且不会被第三方数据提供商内容结构的变化所累。很多新数据都是非结构化或是半结构化的，因此开发者还需要能够高效存储这种数据的数据库。但遗憾的是，关系型数据库所使用的定义严格、基于模式的方式是无法快速容纳新的数据类型的，对于非结构化或是半结构化的数据更是无能为力。NoSQL 提供的数据模型则能很好地满足这种需求。很多应用都会从这种非结构化数据模型中获益，比如说 CRM、ERP、BPM 等等，他们可以通过这种灵活性存储数据而无需修改表或是创建更多的列。这些数据库也非常适合于创建原型或是快速应用，因为这种灵活性使得新特性的开发变得非常容易。

2. NoSQL 很容易实现可伸缩性（向上扩展与水平扩展）

如果有很多用户在频繁且并发地使用你的应用，那么你就需要考虑可伸缩的数据库技术而非传统的 RDBMS 了。对于关系型技术来说，很多应用开发者会发现动态的可伸缩性是难以实现的，这时就应该考虑切换到 NoSQL 数据库上。对于云应用来说，关系型数据库一开始是普遍的选择。然而，在使用过程中却遇到了越来越多的问题，原因就在于他们是中心化的，向上扩展而非水平扩展的。这使得

他们不适合于那些需要简单且动态可伸缩性的应用。NoSQL 数据库从一开始就是分布式、水平扩展的，因此非常适合于互联网应用分布式的特性。

在三层互联网架构的 Web/应用层上，多年来向上扩展已经成为默认的扩展方式了。随着应用使用人数的激增，我们需要添加更多的服务器，性能则是通过负载均衡来实现的，这时的代价与用户数量成线性比例关系。在 NoSQL 数据库之前，数据库层的默认扩展方式就是向上扩展。为了支持更多的并发用户以及存储更多的数据，你需要越来越好的服务器，更好的 CPU、更多的内存、更大的磁盘来维护所有表。然而，好的服务器意味着更加复杂、私有、并且也更加昂贵。这与 Web/应用层所使用的便宜的硬件形成了鲜明的对比。

3.动态模式

关系型数据库需要在添加数据前先定义好模式。比如说，你需要存储客户的电话号码、姓名、地址、城市与州等信息，SQL 数据库需要提前知晓你要存的是什么。这对于敏捷开发模式来说是场灾难，因为每次完成新特性时，数据库的模式通常都需要改变。因此，如果在开发过程中想将客户喜欢的条目加到数据库中，那就得向表中添加这一列才行，然后要做的就是将整个数据库迁移到新的模式上。

4.自动分片

由于是结构化的，关系型数据库通常会垂直扩展，单台服务器要持有整个数据库来确保可靠性与数据的持续可用性。这样做的代价就是非常昂贵、扩展受到限制，并且数据库基础设施会成为失败点。这个问题的解决方案就是水平扩展，添加服务器而不是为单台服务器增加更多的能力。NoSQL 数据库通常都支持自动分片，这意味着他们本质上就会自动在多台服务器上分发数据，应用甚至都不知道这些事情。数据与查询负载会自动在多台服务器上做到平衡，当某台服务器当机时，它能快速且透明地被替换掉。

5.复制

大多数 NoSQL 数据库也支持自动复制，这意味着你可以获得高可用性与灾备恢复功能。从开发者的角度来看，存储环境本质上是虚拟化的。

NoSQL 数据库面临的挑战

1.成熟度

RDBMS 系统由来已久。NoSQL 拥护者们会说 RDBMS 的高龄是其衰退的标志，

不过对于大多数 CIO 来说，RDBMS 的成熟让人放心。对于大多数情况来说，RDBMS 系统是稳定且功能丰富的。相比较而言，大多数 NoSQL 数据库则还有很多特性有待实现。

2. 支持

企业需要的是安心，如果关键系统出现了故障，他们可以获得即时的支持。所有 RDBMS 厂商都在不遗余力地提供良好的企业支持。与之相反，大多数 NoSQL 系统都是开源项目，虽然每种数据库都有那么几家公司提供支持，不过这些公司大多都是小的初创公司，没有全球支持资源，也没有 Oracle、微软或是 IBM 那种令人放心的公信力。

3. 分析与商业智能

NoSQL 数据库在 Web 2.0应用时代开始出现。因此，大多数特性都是面向这些应用的需要的。然而，应用中的数据对于业务来说是有价值的，这种价值远远超出了 Web 应用那种 CRUD。企业数据库中的业务信息可以帮助改进效率并提升竞争力，商业智能对于大中型企业来说是个非常关键的 IT 问题。

4. 管理

NoSQL 的设计目标是提供零管理的解决方案，不过当今的现实却离这个目标还相去甚远。现在的 NoSQL 需要很多技巧才能用好，并且需要不少人力、物力来维护。

5. 专业

全球有很多开发者，每个业务部门都会有熟悉 RDBMS 概念与编程的人。相反，几乎每个 NoSQL 开发者都处于学习模式。这种状况会随着时间的流逝而发生改观。但现在，找到一个有经验的 RDBMS 程序员或是管理员要比 NoSQL 专家容易多了。

结论

NoSQL 数据库正在成为数据库领域的重要力量。如果使用恰当，那么它会带来很多好处。然而，企业应该非常小心并注意到这些数据库的限制与问题。

原文链接：http://www.infoq.com/cn/news/2014/01/nosql-vs-rdbms?utm_source=tuicool

Redis 作者谈 Redis 应用场景

毫无疑问，Redis 开创了一种新的数据存储思路，使用 Redis，我们不用在面对功能单调的数据库时，把精力放在如何把大象放进冰箱这样的问题上，而是利用 Redis 灵活多变的数据结构和数据操作，为不同的大象构建不同的冰箱。希望你喜欢这个比喻。

下面是一篇新鲜出炉的文章,其作者是 Redis 作者@antirez,他描述了 Redis 比较适合的一些应用场景，NoSQLFan 简单列举在这里，供大家一览：

1. 取最新 N 个数据的操作

比如典型的取你网站的最新文章，通过下面方式，我们可以将最新的5000条评论的 ID 放在 Redis 的 List 集合中，并将超出集合部分从数据库获取

□ 使用 LPUSH latest.comments<ID>命令，向 list 集合中插入数据

□ 插入完成后再用 LTRIM latest.comments 0 5000命令使其永远只保存最近5000个 ID

□ 然后我们在客户端获取某一

```
FUNCTION get_latest_comments(start,num_items):
    id_list = redis.lrange("latest.comments",start,start+num_items-1)
    IF id_list.length < num_items
        id_list = SQL_DB("SELECT ... ORDER BY time LIMIT ...")
    END
    RETURN id_list
```

END 如果你还有不同的筛选维度，比如某个分类的最新 N 条，那么你可以再建一个按此分类的 List，只存 ID 的话，Redis 是非常高效的。

2. 排行榜应用，取 TOP N 操作

这个需求与上面需求的不同之处在于，前面操作以时间为权重，这个是以某个条件为权重，比如按顶的次数排序，这时候就需要我们的 sorted set 出马了，将你要排序的值设置成 sorted set 的 score，将具体的数据设置成相应的 value，每次只需要执行一条 ZADD 命令即可。

3. 需要精准设定过期时间的应用

比如你可以把上面说到的 sorted set 的 score 值设置成过期时间的时间戳，那么就可以简单地通过过期时间排序，定时清除过期数据了，不仅是清除 Redis 中的过期数据，你完全可以把 Redis 里这个过期时间当成是对数据库中数据的索引，用 Redis 来找出哪些数据需要过期删除，然后再精准地从数据库中删除相应的记录。

4. 计数器应用

Redis 的命令都是原子性的，你可以轻松地利用 INCR，DECR 命令来构建计数器系统。

5. Uniq 操作，获取某段时间所有数据排重值

这个使用 Redis 的 set 数据结构最合适了，只需要不断地将数据往 set 中扔就行了，set 意为集合，所以会自动排重。

6. 实时系统，反垃圾系统

通过上面说到的 set 功能，你可以知道一个终端用户是否进行了某个操作，可以找到其操作的集合并进行分析统计对比等。没有做不到，只有想不到。

7. Pub/Sub 构建实时消息系统

Redis 的 Pub/Sub 系统可以构建实时的消息系统，比如很多用 Pub/Sub 构建的实时聊天系统的例子。

8. 构建队列系统

使用 list 可以构建队列系统，使用 sorted set 甚至可以构建有优先级的队列系统。

9. 缓存

这个不必说了，性能优于 Memcached，数据结构更多样化。

原文链接：http://blogread.cn/it/article/3939?f=hot1&utm_source=tuicool

SQL 语句的 LIMIT 的用法

SELECT * FROM table LIMIT [offset,] rows | rows OFFSET offset

mysql> SELECT * FROM table LIMIT 5,10; // 检索记录行 6-15

//为了检索从某一个偏移量到记录集的结束所有的记录行,可以指定第二个参数为 -1：

mysql> SELECT * FROM table LIMIT 95,-1; // 检索记录行 96-last.

//如果只给定一个参数，它表示返回最大的记录行数目：

mysql> SELECT * FROM table LIMIT 5; //检索前 5 个记录行

//换句话说，LIMIT n 等价于 LIMIT 0,n。select * from table LIMIT 5,10; #返回第6-15行数据

select * from table LIMIT 5; #返回前5行

select * from table LIMIT 0,5; #返回前5行

1、offset 比较小的时候。 select * fromyanxue8_visit limit 10,10

多次运行，时间保持在0.0004-0.0005之间

Select * From yanxue8_visit Where vid >=(Select vid From yanxue8_visit Order By vid limit 10,1) limit 10

多次运行，时间保持在0.0005-0.0006之间，主要是0.0006 结论：偏移offset 较小的时候，直接使用 limit 较优。这个显然是子查询的原因。

2、offset 大的时候。 select * from yanxue8_visit limit 10000,10 多次运行，时间保持在0.0187左右

Select * From yanxue8_visit Where vid >=(Select vid From yanxue8_visit Order By vid limit 10000,1) limit 10 多次运行，时间保持在0.0061左右，只有前者的1/3。可以预计 offset 越大，后者越优。

性能优化：

基于 MySQL5.0中 limit 的高性能,我对数据分页也重新有了新的认识.

Select * From cyclopedia Where ID>=(

Select Max(ID) From (

Select ID From cyclopedia Order By ID limit90001

```
) As tmp
) limit 100;
```

2.

```
Select * From cyclopedia Where ID>=(
Select Max(ID) From (
 Select ID From cyclopedia Order By ID limit90000,1
) As tmp
) limit 100;
```

同样是取90000条后100条记录,第1句快还是第2句快?

第1句是先取了前90001条记录,取其中最大一个 ID 值作为起始标识,然后利用它可以快速定位下100条记录

第2句择是仅仅取90000条记录后1条,然后取 ID 值作起始标识定位下100条记录

第1句执行结果.100 rows in set (0.23) sec

第2句执行结果.100 rows in set (0.19) sec

很明显第2句胜出.看来 limit 好像并不完全像我之前想象的那样做全表扫描返回 limitoffset+length 条记录,这样看来 limit 比起 MS-SQL 的 Top 性能还是要提高不少的.

其实第2句完全可以简化成

```
Select * From cyclopedia Where ID>=(
Select ID From cyclopedia limit 90000,1
)limit 100;
```

直接利用第90000条记录的 ID,不用经过 Max 运算,这样做理论上效率因该高一些,但在实际使用中几乎看不到效果,因为本身定位 ID 返回的就是1条记录,Max 几乎不用运作就能得到结果,但这样写更清淅明朗,省去了画蛇那一足.

可是,既然 MySQL 有 limit 可以直接控制取出记录的位置,为什么不干脆用 Select * From cyclopedia limit90000,1呢?岂不更简洁?

这 样想就错了,试了就知道,结果是:1 row in set (8.88)sec,怎么样,够吓人的吧,让我想起了昨天在4.1中比这还有过之的"高分".Select *最好不要随便用,要本着用什么,选什么的原则, Select 的字段越多,字段数据量越大,速度就越慢.

上面2种分页方式哪种都比单写这1句强多了,虽然看起来好像查询的次数更多一些,但实际上是以较小的代价换取了高效的性能,是非常值得的.

第1种方案同样可用于 MS-SQL,而且可能是最好的.因为靠主键 ID 来定位起始段总是最快的.

```
 Select Top 100 * From cyclopedia Where ID>=(
Select Top 90001 Max(ID) From (
 Select ID From cyclopedia Order By ID
) As tmp
```

但不管是实现方式是存贮过程还是直接代码中,瓶颈始终在于 MS-SQL 的 TOP 总是要返回前 N 个记录,这种情况在数据量不大时感受不深,但如果成百上千万,效率肯定会低下的.相比之下 MySQL 的 limit 就有优势的多,执行:

```
Select ID From cyclopedia limit 90000
Select ID From cyclopedia limit 90000,1
```

的结果分别是:

```
90000 rows in set (0.36) sec
1 row in set (0.06) sec
```

而 MS-SQL 只能用 Select Top 90000 ID From cyclopedia 执行时间是390ms,执行同样的操作时间也不及 MySQL 的360ms——LIMIT 思考 PERCONA PERFORMANCE CONFERENCE 2009上，来自雅虎的几位工程师带来了一篇”EfficientPagination Using MySQL“的报告，有很多亮点，本文是在原文基础上的进一步延伸。首先看一下分页的基本原理：mysql> explain SELECT * FROM message ORDER BY id DESC LIMIT 10000, 20G

```
*************************** 1. row ***************
id: 1
select_type: SIMPLE
table: message
type: index
possible_keys: NULL
key: PRIMARY
```

key_len: 4

ref: NULL

rows: 10020

Extra:

1 row in set (0.00 sec)limit10000,20的意思扫描满足条件的10020行，扔掉前面的10000行，返回最后的20行，问题就在这里，如果是 limit100000,100，需要扫描100100行，在一个高并发的应用里，每次查询需要扫描超过10W 行，性能肯定大打折扣。文中还提到 limitn 性能是没问题的，因为只扫描 n 行。文中提到一种”clue”的做法，给翻页提供一些”线索”，比如还是 SELECT * FROM message ORDER BYidDESC，按 id 降序分页，每页20条，当前是第10页，当前页条目 id 最大的是9527，最小的是9500，如果我们只提供”上一页”、”下一页”这样的跳转（不提供到第 N 页的跳转），那么在处理”上一页”的时候 SQL 语句可以是：SELECT * FROM message WHERE id > 9527 ORDER BYid ASC LIMIT 20;处理”下一页”的时候 SQL 语句可以是：SELECT * FROM message WHERE id < 9500 ORDER BYid DESC LIMIT 20;不管翻多少页，每次查询只扫描20行。缺点是只能提供”上一页”、”下一页”的链接形式，但是我们的产品经理非常喜欢”<上一页 1 23 4 5 6 7 8 9 下一页>”这样的链接方式，怎么办呢？如果 LIMIT m,n 不可避免的话，要优化效率，只有尽可能的让 m 小一下，我们扩展前面的”clue”做法，还是 SELECT *FROM message ORDER BY idDESC，按 id 降序分页，每页20条，当前是第10页，当前页条目 id 最大的是9527，最小的是9500，比如要跳到第8页，我看的 SQL 语句可以这样写：SELECT * FROM message WHERE id > 9527 ORDER BYid ASC LIMIT 20,20;跳转到第13页：SELECT * FROM message WHERE id < 9500 ORDER BYid DESC LIMIT 40,20;原理还是一样，记录住当前页 id 的最大值和最小值，计算跳转页面和当前页相对偏移，由于页面相近，这个偏移量不会很大，这样的话 m 值相对较小，大大减少扫描的行数。其实传统的 limitm,n，相对的偏移一直是第一页，这样的话越翻到后面，效率越差，而上面给出的方法就没有这样的问题。注意 SQL 语句里面的 ASC 和 DESC，如果是 ASC 取出来的结果，显示的时候记得倒置一下。已在60W 数据总量的表中测试，效果非常明显。

原文：http://www.cnblogs.com/wangxingliu/p/3512188.html?utm_source=tuicool

MapReduce 编程模型

MapReduce 是一个 Google 发明的编程模型，也是一个处理和生成超大规模数据集的算法模型的相关实现。用户首先创建一个 Map 函数处理一个基于<Key,Value>对的数据集合，输出的中间结果基于<k,v>对的数据集合，然后再创建一个 Reduce 函数用来合并所有的具有相同中间 Key 值的中间 Value 值。

MapReduce 架构的程序可以实现在大量普通配置的设备上实现分布式计算。在 Google 的集群中，每天都有1000多个 MapReduce 程序在执行。

例如下面的例子：计算系一个大的文档集合中每个单词出现的次数，伪代码如下：（代码来自 Google 的论文）

[plain] view plaincopyprint?

```
map(String key, String value):
    //key: document name
    //value: document contents
    for each word w in value
        EmitIntermediate(w,"1");

reduce(String key, Iterator values):
    //key: a word
    //values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

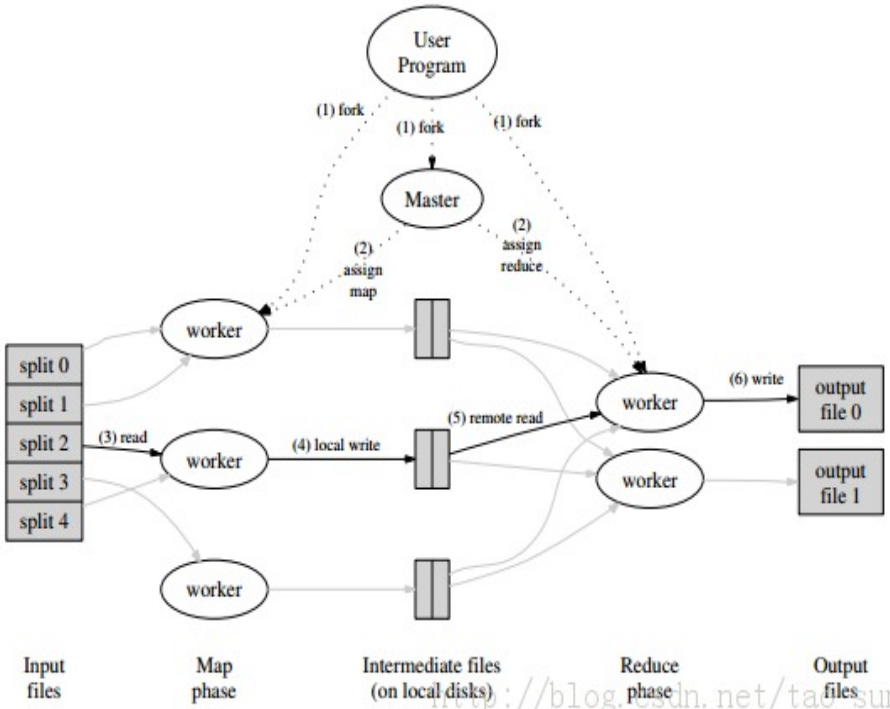
Map 函数输出文档中的每个单词、以及这个词的出现次数。Reduce 函数把 Map 函数产生的每一个特定的词的计数累加起来。

MapReduce 的执行概括：

通过将 Map 调用的输入数据自动分割为 M 个数据片断的集合，Map 调用被分布到多台机器上执行。输入的数据片断能够在不同的机器上并行处理。使用分区

函数将Map调用产生的中间key值分成R个不同的分区(例如,hash(key)mod R),Reduce调用好也被分布到多台机器上执行。分区数量R和分区函数由用户来指定。

下图展示了MapReduce的执行流程:



1）用户程序首先调用MapReduce库将将输入文件分成M个数据片断，每个数据片断的大小一般从16MB到64MB。然后用户程序在集群中创建大量的程序副本。

2）程序副本包括一个Master程序，若干个Worker程序，master负责分配任务，M个map任务，和R个reduce任务。master将一个Map任务或Reduce任务分配给一个空闲的worker。

3)worker程序收到分配到的map任务后将读取相关的数据片断,解析出<k,v>对，然后传递给用户定义的map函数，由map函数生成中间的<k,v>对，并缓存在内存中。

4）缓存中的<k,v>对通过分区函数分成R个区域，然后周期性的写入到本地磁盘中。并且缓存中的<k,v>对在本地磁盘上的存储位置将被回传给master，由master再负责把这些存储位置传送给负责reduce任务的worker们。

5)但负责reduce的worker们收到master程序送来的数据存储位置信息后，

使用 RPC（远程程序调用）从负责 map 的 woker 们所在的主机上的磁盘里读取缓存数据。负责 reduce 的 worker 们得到中间数据后，对 key 进行排序后将具有相同 key 值的数据聚合在一起。

6)然后 reduce worker 将按 key 将中间 value 值传递给用户自定义的 reduce 函数。reduce 函数的输出被追加到所属分区的输出文件。

7）当所有的 map 和 reduce 任务都完成后，master 唤醒用户程序。这时，用户程序对 MapReduce 的调用将返回到用户代码中。

计数器

MapReduce 库使用计数器统计不同事件的发生次数。例如，用户可能想统计已经处理了多少个单词,已经索引了多少篇 German 文章等。为了使用这个特性，用户在程序中创建一个命名的计数器对象，在 map 和 reduce 函数中相应的增加计数器的值。例如：

[plain] view plaincopyprint?

```
Counter* uppercase;
uppercase = getCounter("uppercase");

map(String name, String contents):
    for each word w in contents:
        if (IsCapitalized(w))
            uppercase->Increment();
        EmitIntermediate(w,"1");

```

计数器的值周期性的从各个单独的 worker 机器上传递给 master。master 把执行成功的 Map 和 Reduce 任务的计数器值进行累加，但 MapReduce 操作完成后，返回给用户代码。计数器对于 MapReduce 操作的完整性检查非常有用。

原文链接：

http://blog.csdn.net/tao_sun/article/details/17959405?utm_source=tuicool

【科技英雄传】C++之父：将工作视为一种乐趣

本贾尼-斯特劳斯特卢普（Bjarne Stroustrup）1950年出生于丹麦，先后毕业于丹麦阿鲁斯大学和英国剑桥大学。在完成学业后，斯特劳斯特卢普曾任 AT&T 大规模程序设计研究部门负责人，AT&T、贝尔实验室和 ACM 成员，现任德州农工大学计算机系首席教授。

说起斯特劳斯特卢普，我们不得不提的就是他在1979年开发出的一种在当时被称为“C with Classes”的计算机编程语言，而这一语言便是如今我们所熟知的 C++。

简单来说，所谓 C++指的是一种使用非常广泛的计算机编程语言，该语言是一种静态数据类型检查、支持多重编程范式的通用程序设计语言。而且，C++的编译器比目前其他计算机语言的编译技术更复杂。

在计算机诞生初期，人们要使用计算机必须用机器语言或汇编语言编写程序。世界上第一种计算机高级语言“FORTRAN”诞生于1954年，随后还先后出现了多种计算机高级语言。其中使用最广泛、影响最大的无疑是 BASIC 和 C 语言。

BASIC 语言是1964年在 FORTRAN 语言的基础上简化而成的，它是为初学者设计的小型高级语言。C 语言则是1972年由美国贝尔实验室的 D.M.Ritchie 推出，它不是为初学者设计的，而是主要为计算机专业人员设计。

在当时，大多数系统软件和许多应用软件都是用 C 语言编写的，但是随着软件规模的不断扩大，用 C 语言编写程序的短板已经愈发明显。因此在 C 基础上，斯特劳斯特卢普进一步扩充和完善了 C 语言的不足而开发出了 C++语言。

据斯特劳斯特卢普自己透露称，当时他正在负责一个软件项目，但那时没有任何一种计算机语言能够满足自己复杂的工作需求，所以斯特劳斯特卢普才决定在 C 语言的基础上逐步对其进行改进。

1985年，C++语言被正式定义，外界将其视为比 C 语言更加高效的计算机编程语言。1998年，ANSI/ISO C++标准建立，斯特劳斯特卢普也在同年推出了经典

著作《The C++ Programming Language》第三版，因而他本人被尊称为“C++语言之父”。

事实上，在C++语言诞生的道路上还出现了一个小插曲。因为斯特劳斯特卢普此前一直将这一语言称作“C with Classes”，直到1983年12月他才采纳了同行里克-马克西帝（Rick Mascitti）的建议，将自己发明的新语言命名为更为简洁的“C++”。

“软件行业太多的经理和管理人员试图把编程变成低级别的流水线工作，从长远来看这种做法效率低、浪费大、成本昂贵，且非常不人性化。在软件开发领域，没有放之四海而皆准的模型，因此需要给予人们充分的发挥空间。”斯特劳斯特卢普在接受IBM发明大师、DB2产品开发团队研发经理及高级技术人员萨姆-莱特斯通（Sam Lightstone）采访时说道。

目前，斯特劳斯特卢普在闲暇时候的最大乐趣便是同家人、朋友在一起、出门旅游、拍照、听音乐，但他认为自己所负责的部分项目本身也十分有趣。

“我简直不敢想象干这么有趣的事还能拿薪水。”斯特劳斯特卢普最后说道。

原文链接：http://tech.qq.com/a/20140109/000296.htm?utm_source=tuicool

从《安德的游戏》看如何与外星人沟通

在奥森·斯科特·卡德（Orson Scott Card）的著名科幻小说和改编电影《安德的游戏》中，核心冲突之一是人类无法和虫族沟通，而这一事实成为了人类和

虫族开战的根本理由。按照小说的设定，虫族用思维交流，因而没有语言、没有阅读写作、没有信号、没有数字，甚至没有通讯设备接受我们的信号，所以我们没法交流——除非我们也找到了心灵感应能力。

但是细想一下，这其实有点说不通。就算虫族个体之间能够完全靠心灵沟通，它们总要感知外界环境，总得有视力和听力吧？而且虫族只要是演化来的，就不可能是母星上的唯一生命形态，它们和其它物种要如何心灵感应？为了存活，它们至少得能做出一些原始的恐吓动作或者声音，而且要能理解对方的恐吓姿态，这就意味着它们对外不可能是石头一块。最重要的是，按照小说的描述，虫族是技术文明，它们不是血肉之躯飞过来，而是驾驶员开着飞船前来入侵的，要驾驶飞船难道不需要机载电脑，不用读仪表盘吗？不论怎么想，开发一种基于视觉的沟通途径至少都值得尝试一下。

那么，面对初次相遇的异形文明，我们要怎么让它们知道我们也是智慧生命，又怎么和它们建立交流呢？

数学/科学语言

纯数学家和逻辑学家其实一直有一个梦想：创造一种完全精确、完全符合逻辑、毫无歧义的语言。如果用这种语言来讨论那些最艰深的问题，就不会因为语言而陷入任何误解或逻辑陷阱了。显然这个梦想还没有实现，但是不妨碍他们去尝试创造这种语言——而这样的东西用来和外星人沟通，应该是最合适不过了吧！别的东西再怎么变，逻辑总归是永存的，一个东西不可能既是 A 又是非 A。

1960年，荷兰天文学家汉斯·弗勒登塞尔（Hans Freudenthal）基于这种思路设计了“宇宙语”（Lincos，是拉丁语 lingua cosmica 的缩写）。他的想法是，第一部分用最简单的脉冲确立二进制的数字和基本四则运算（4个脉冲 ">" 2个脉冲， 3个脉冲 “+” 5个脉冲 “=” 8个脉冲，外星人应该能很快猜到> 、 + 、 = 都是什么意思），然后用一系列的例子来阐述相等、比较、变量、常量等概念，然后是形式逻辑和集合论。第二部分用来讲解时间、度量、过去和未来的概念。第三部分用于阐述如何描述个体间的交流，如何表达“A 向 B 提问”、“B 不赞同 A”、“C 引用了 B 的话”等状态。第四部分用来讲解物理，包括质量、运动和空间，顺带简单介绍人类的物理特性和太阳系的基本信息。

以下是第三部分的宇宙语伪代码实例之一，大家可以试试猜测这是在干什么，

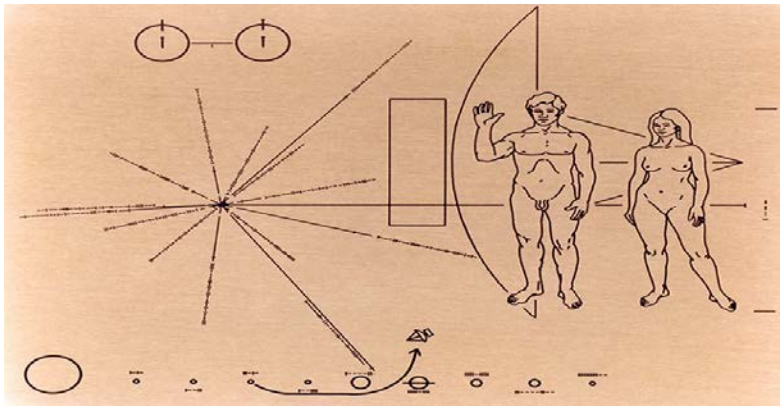
每个“词”又是什么意思。

```
    Ha Inq Hb ?x 2x=5
Hb Inq Ha 5/2
Ha Inq Hb Ben
Ha Inq Hb ?x 4x=10
Hb Inq Ha 10/4
Ha Inq Hb Mal
Hb Inq Ha 1/4
Ha Inq Hb Mal
Hb Inq Ha 5/2
Ha Inq Hb Ben
Hc Inq Hd ?y y Inq Hb ?x 4x=10
Hd Inq Hc Ha
```

当出现通讯可能时，我们就把以上这本“字典”发送过去。作者本来计划了第二本书，来讲物质、地球、生命和行为，但是没有写出来……

卡尔·萨根（Carl Sagan）在他的著名科幻小说《接触》（已改编为电影，电影译名《超时空接触》）里，就提到了“宇宙语”的一个变体。还有好几位研究者在此基础上进行了改良。不过直到1999年，才有人用改良过的宇宙语编码向邻近天体发送信息。幸运（或不幸）的是，迄今为止，我们还没有收到回复。

图像



先驱者10号和11号携带的图像铭牌。

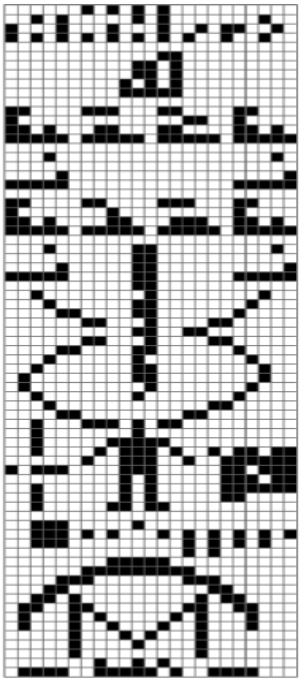
语言设计得再巧妙，毕竟是个抽象的东西。所谓一图胜千言，我们为啥不直接弄图呢？的确，缺乏编码协议，传输一个 gif 动图不太可能，但 bmp 点阵总归

是可以做到的吧……

这个思路粗看起来比语言靠谱得多，因为不需要从底层逻辑一步步建起，就算对方没收到完整信息也没关系。所以，先驱者10号和11号各带了一块图像铭牌，旅行者带的金色唱片表面上也刻了图像。

但是实体毕竟受限太大，所以研究者设计了著名的“阿雷西博信号”，于1974年11月16日发射出去。它有意设计成了1679个二进制位，可以唯一分解成73＊23，用来提示接受者这是一个73行、每行23位的二维矩阵（如果接受者改用23*73，会发现图像变得杂乱无章）。

设计者宣称在短短的210字节里植入了大量信息：1到10的数字，氢碳氮氧磷的原子序数，DNA 的基本化学成分，双螺旋的形状，人的形状和高度以及地球人口，太阳系示意图，阿雷西博望远镜的示意图和尺寸，等等…… 但坦率地说，我认为外星人不可能看懂这么多东西。大家还是自行判断一下吧。（参见：外星人，你能看懂我们发给你的信息吗？）



阿雷西博信号的像素图转码。

此外，阿雷西博信号要花25000年才能抵达目的地，而且送到时，它的目的地——M13——已经不在那里了。所以这次信号发放没啥实际意义，只是个演示。

算法

我们知道，算法如果使用得当，就是有史以来最高效的压缩手段。最极端的

例子里，一个π的生成算法可以产生出任意多位的π近似值。如果直接送去信息有困难，我们是不是可以送去一个程序呢？这是比“只送大脑”更疯狂的想法，但是出乎意料地，似乎是可行的。

譬如美国人保罗·菲兹派翠克(Paul Fitzpatrick)就设计了这样一个语言，名为“CosmicOS”。它以Lincos为基础，从Scheme（一种Lisp的方言）里吸取了极简主义灵感，只用4个不同的符号——0、1、左括号、右括号。数字就是一对括号里的一个二进制串，运算符就是程序自己定义好的数字。基于这种语言，作者设计了一套可执行的开源代码，大家可以去sourceforge围观。

但是，要如何编码它并广播出去，还没有达成一致意见，所以它还没有投入使用……

当然了，我们也可以把以上几种都综合起来，比如乌克兰叶夫帕托里亚天文台就从1999年到2003年间发射了名为“宇宙呼唤”的综合信号，把文字、图片、声音、影像、宇宙语、代码之类的打包一起发出去了，还发给了好多不同的恒星系统，最早一个会在2036年抵达。这可真是作大死的节奏呀。

所以，回到开篇的问题。面对《安德的游戏》中的虫族，不靠心灵感应的话，要怎样才能沟通？

小说设定虫族飞船不靠无线电。但是很难想象没有射电望远镜，它们要如何探索宇宙，如何准确判断适合殖民的星球方位。所以在它们的母星上很可能至少有一个无线电接收器——我们如果向那里发送信息的话，它们原则上可以收到，而且足够聪明的话应该是可以解开的。

如果不用无线电，其实还可以直接视觉交流。在小说中，虫族被设定为有中心大脑的集合个体——母后充当大脑，而工蚁相当于它的肢体，之间靠心灵感应沟通。工蚁必须有感官才能回馈大脑，让大脑知道如何应对，所以我们抓一个俘虏，给它看图像，就等于给它的母后看图像了。这好像比个体物种还要容易一些……

当然，要想实现以上的梦想，首先，你要，有个，外星人。费米悖论的解释之一就是，外星人确实不存在，或者存在的时间太短，以至于我们实际上无法交流…… 虽然这种前景比较黯淡，但至少比被入侵或者被清理要强一点儿。

原文链接：http://www.guokr.com/article/437862/?utm_source=tuicool

专访何海涛：“不正经”程序员的进阶之路

个人成长经历

CSDN：请和大家介绍下你及目前所从事的工作。

何海涛：我目前在微软上海的 CCIC（China Cloud Innovation Center）组从事软件开发工作，项目与微软的 Windows Azure 云计算服务在中国落地相关，需要的技术除了熟悉 Windows Azure 平台外，还需要熟悉 ASP .NET 以及网页开发技术比如 JavaScript 等。

CSDN：在大学时，你曾是党支部书记、团支部书记、班长、计算机学院学生会副主席、计算机学院研究生会副主席等职务，作为一名学生，你是如何权衡学习和学生工作之间关系的？这对你进入到社会后的工作有什么影响？

何海涛：学习对于学生来说是最重要的。对于计算机相关专业的学生来说，除了学好书本知识，更重要的是提高编程技术。如果还有时间和精力，多参加社团活动对今后的发展有好处。我说的社团包括具有一定官方性质的比如学生会这样的社团，也包括非官方的文体兴趣方面的社团。大家可以根据自己的兴趣选择适合自己的社团。

对大多数同学而言，可能是把过多的时间投入到比如打游戏、上网看视频这些休闲活动中去了，并不是真的没有时间去参加各种社会活动。如果能把部分休闲时间用到参加社会活动中来，相信会有所收获。

我学生时代参加过不少社团活动，使我的交流沟通能力得到了很大的提高。**我对自己的要求是坐下来能写，站起来能说。**我的口头沟通能力和笔头的沟通能力都是在大学时代得到了锻炼。这几年我写书、去高校演讲，都得益于这些能力的积累。目前由于兴趣的原因，我还没有转到管理职位上去。但我相信自己之前积累的一些管理沟通的技能，能够胜任管理岗位的需要。

CSDN：你在去年每天陪小孩玩的时间都在三小时以上，此前的情况是如何的呢？作为一名程序员处理工作和生活之间关系方面，你有什么经验可分享？

何海涛：13年我能有大量时间陪伴家人，是由于身体原因有意放慢了生活的节奏。和家人聊天、陪小孩做游戏，能感受到极大的幸福。

我11年和12年相对而言要忙一些，那两年除了工作之外写了两本书。当需要

在家里工作的时候，先和家人做好沟通并得到他们的支持。在写书的那段时间，我会跟老婆说今天晚上9点到11点我要写书，不能陪你。然后我会尽量高效率的完成计划的进度。事情都按着计划的进度完成，反过来也就能抽出时间陪伴家人。**我特别自豪的一件事情是11年老婆怀孕到12年3月份小孩出生，我每天晚饭后都陪她散步爬楼梯，没有中断过一天。也正在那段时间我完成了第一本书的写作、改稿。**

有点“不正经”的程序员——把兴趣和职业分开

CSDN：你在 Autodesk 工作期间，曾参加公司 AutoLove 自愿者活动，前往贵州三都水族自治县水龙乡小学做志愿者老师。可能很多人都有过此类的想法，但是做的人寥寥无几，行大于言的你值得人敬佩，能否简单介绍这次支教活动以及参加的原因？这次支教经历你最大的感悟是什么？

何海涛：AutoLove 是 Autodesk 的一个公益项目。Autodesk 在贵州三都水族自治县水龙乡小学捐助了一幢宿舍楼，并每年都选5名志愿者去学校支教一个星期。我是农村长大的孩子。当年自己渴望知道外面的世界但条件很有限。所以在07年我知道公司有这个活动之后就报名了。记得我还准备了一节课，是专门介绍各大城市的特点的。

参加一次活动是容易的，但坚持和学校的老师和同学保持联系，却很难做到。这一点我自己也做得不够好。从贵州回来之后，还有几个孩子给我写信。但我只坚持写了两年回信，后来也逐渐断了联系。

CSDN：你的“不正经”可能不仅在于变身支教老师，你还对财经管理等领域颇为关注，曾担任了《头脑风暴》栏目（在第一财经以及宁夏卫视播出）的嘉宾。这一次你作何解释呢？程序员在大家的印象中是不善言辞、死板的印象，对与改变这一现状此你有什么经验可分享？

何海涛：我比较喜欢阅读，经常看一些财经管理类的文章和书籍。同时作为一个不太成功的股票投资者，也时常关注财经新闻。所以财经管理的知识不算匮乏。

选择程序员作为职业的同行，绝大多数的智商都很高，要学什么都很快。很多人也有自己的兴趣爱好，并且钻研得很深。我就有很多同事的摄影技术相当专业。只是有不少人喜欢自娱自乐，不太喜欢和其他人交流。

在技术方面也一样，有不少技术功底很扎实的程序员只喜欢埋头干活，不太喜欢抬头和他人交流。其实要成为一个有影响力的程序员，和他人交流的能力很重要。写博客，在行业大会上演讲，这些都是扩大自己影响力的有效方法。所以我觉得 **要成为一个真正有影响力的程序员，既能写出好的代码，也能把好的技术、方法和理念传播给更多的人。**

CSDN：仅仅凭借以上两点可能说你“不正经”有点理由不足。你近年来还积极参与 JA（Junior Achievement，青年成就），多次去高校和大学生分享关于职业规划以及建立人际关系网络等的心得和经验，俨然化身为了一名讲师，总结多年的讲课经历，大学生在职业规划和建立人际关系网方面都存在哪些问题？

何海涛：这几年“宅”字很流行，这也说明有很多年轻人习惯生活在一个狭小的世界里。很多学生在网络或者游戏等虚拟世界里投入了过多的时间，但忽视了身边的同学朋友。他们还没有意识到他们身边的同学、师兄师姐、朋友甚至朋友的朋友，都有可能是今后职业发展的重要资源。因此在做 JA 讲师的时候， **我会鼓励大学生朋友能通过各种途径参与社会活动，建立人际关系网，为今后的职业发展积累资源。**

CSDN：以为就这样了嘛，还不够，在11年期间你又出版了《剑指 Offer——名企面试官精讲典型编程题》一书，写书不是一朝一夕的事情，你肯定花费了大量的时间和精力，程序员该如何利用好自己工作之外的时间呢？

何海涛：我曾在 《程序员》杂志上写过一篇短文小结自己在写《剑指 Offer》期间时间管理的心得——善用时间，发展副业。主要有以下四点：

- 明确任务与目标
- 细化目标，执行计划
- 善用闲散时间
- 区分使用小段时间和大段时间

一旦下定决心完成某一任务，就要在确定总体目标之后细化每一周甚至每一天的工作量，尽量确保每个时间节点都能按时完成。由于副业挤占了休息时间，可能会很累。虽然不是说完全不能有休闲活动，但如果有可能还是可以把闲散时间利用起来，做一些工作相关的事情，这样既放松了大脑，又有效利用了时间。

如果能坚持一段时间，就会小有所成。

CSDN：支教老师、电视台节目嘉宾、大学生讲师、著书人等，对于一个工作仅七年的程序员来说似乎是“不正经”，可你看起来都把每一件事情做得很多，比很多“正经”了很多，回顾这些事情，对你都有哪些的帮助？是什么让你一路坚持走来的？

何海涛：我是一个兴趣比较广泛的人，很多事情都有兴趣去尝试一下。这样我的人生才不算单调。之前我也曾考虑过是不是可以全职写书。后来我跟太太有过深入的探讨，她建议我 **要把兴趣和职业分开** 。这个建议现在看来很有道理。我现在有一个自己喜欢的职业，在做了七年程序员之后还喜欢每天都写代码。同时尝试一些有意思的事情，让自己的生活保持新鲜感。

原文链接：http://www.csdn.net/article/2014-01-08/2818061?utm_source=tuicool

日记——程序员的烦恼

前天（2014年1月7日，阴，据说有人在延庆看到了几片雪花）

我用了一晚上的时间写了三千字，写完后已至凌晨。古人云，白发三千丈，缘愁似个长。写完三千字之后我找到了这种感觉，又读了 N 遍，改了 N 遍，终于沉沉睡去。醒来后再读，发现自己被感动了。这一点充分证实了，Mac 君依然在写作的道路上狂飙突进，以一个二把刀的身份。希望在下个阶段，Mac 君能够

从二把刀转变为一个合格的工匠。

这三千字，暂时还不能发。

昨天（2014年1月8日，晴，大风吹）

我开了一天的会，会议从早上9点持续到晚上7点，期间吃了一碗面。参会的每个人似乎都发言了，但我忘了他们说了些什么。我自己也发言了，我也忘记了自己说了些什么。似乎做了很多决定和决策，又似乎什么都没定下来。我很疲惫。晚上回到家想写点什么，打开微信后台看到有个叫麻花的兄弟在吟唱：强哥，发个文章看看吧，火车上睡不着啊。我心想，难道睡不着也要强哥哄吗？多大岁数了都！接着我又打开了用户分析，取消关注的人数又增加了几十个，于是我很伤感。掐指算算，这一年取消关注的读者也有2万了，好在新增人数永远大于取消关注人数，否则后果是严重的，成为负数也是可能的。据说防止取消关注的秘诀就是什么都不写，让你的微信号静静的躺在读者的折叠文件夹里，既不发声，也不回复，也许会躲过一劫。于是我决定这个晚上什么都不写……然后眼前一黑，睡过去了。

今天（2014年1月9日，晴的厉害，气温似乎更低了）

事情似乎都走上了正轨，下午一口气处理完这几天累积的「年关例行事务」，效率还不错，看来从明天开始就可以写点代码并考虑2014年产品规划和实施的事情了。2013年我们的平台产品做了一个重要的 Release，开发环境大部分都迁移到了 Git 和 Maven 上，如果没什么变化的话，2014年我会把重点放到企业移动应用和 UI 模式优化上。要做的工作还很多，不过我不急，只要有人、时间和空间，事情总是可以做成的。

前几天送书活动公开了我的 icloud 邮箱，最近收到了不少读者来信，其中大部分都属于「程序员的烦恼」，今天有点时间，做个答读者问吧。

1、工作中提前完成了自己的任务，应该学习提升还是去帮助他人？

帮助别人完成任务也是任务。如果你提前完成了自己的工作，最好的做法是去问你的上级接下来的任务是什么。如果暂时没有其他任务也没有兄弟找你帮忙解决问题，那么就去看书学习吧。提升自己是利人利己的事。

另外，不要忘了好好利用剩余的8小时。

2、应届毕业生投简历到 BAT（百度阿里腾讯）石沉大海怎么破？

那就不要投嘛。据我了解，BAT 对应届毕业生的要求还是很高的，候选人太多，自然可以好好挑挑。石沉大海并不意味着你不够优秀，只是你现在不够优秀而已。放低身段，先去一些不知名的公司锤炼一下，如果能够做到独当一面，你的眼光和技能、经验都会不一样的，到时候再「说英雄谁是英雄」。

3、我现在从事 Android 开发，但我看好 iOS，我应该放弃 Android 转投 iOS 的怀抱吗？

如果你在 Android 的怀抱里获得的是冰冷，在 iOS 那也未必能找到温暖。Android-Java，iOS-Objective-C，虽然我更偏爱 iOS，但是这两个平台和技术，都是好技术，要不然怎么能分庭抗礼这么多年呢？如果你觉得做 Android 悲催，其实是你自己悲催，任何一门技术做到顶尖都是需要坚持和磨练的。

另外，为神马不双飞呢？这又不是结婚，亲吻 Android 的同时，不妨碍你拉起 iOS 的小手啊。少刷微博少看电视，就行了。

4、操作系统、算法、数据结构、设计模式到底该不该学，工作中根本用不到嘛！

最近搞「自媒体」的人似乎功利心大盛，每个人都觉得自己写了这么多字，我容易吗？也该有回报了吧？也该套现了吧？所以好多事情就有些变样。还好我不是自媒体，不用担心这个。

其实学习也是一样，不要那么功利。如果每学一门知识都想着怎么换成银元，第一是心累，第二是学成二把刀的几率大大增加。做为一个程序员，操作系统、算法、数据结构、设计模式等基础知识当然应该学，这是程序员的尊严。至于实际工作中是否能用的上，那是另一回事，用上了就是你的运气，用不上也是很自然的，我小时候还学了一大堆无线电知识呢，最终也没成为无线电专家啊。有人说很多技术不用就会忘记了，那就用呗，做开源项目，写文章，这都是用。退一步就算忘了也没什么，再看一遍就是了。

多少美好的事物，都他妈的毁在了功利二字上！

原文链接：http://macshuo.com/?p=998&utm_source=tuicool

程序员的“横向发展”

在我小的时候，家长经常对胖孩子打趣说：哟，身体长得挺快，可惜就是横向发展了。看来在很多人的潜意识里，纵向发展是向上的，值得夸奖，横向发展则不是那么光彩的事情。但是我的工作经历和思考，却让我对“横向发展”有了新的认识。

程序员的发展，长期以来都是大家关心的问题。通常程序员的发展有两大方向，深度和广度。深度发展，就是精深自己的本事，研习新潮尖端的技术乃至学会“屠龙之术”，以绝招打遍天下；广度发展，就是拓宽自己的技能种类，比如学会更多的语言，以完成更多种类的任务。除去这两大方向，其它能选的发展方向似乎就只有“改行”了。

今天我要说的当然不是改行，而是除去深度发展、广度发展之外的第三维度，因为似乎一直也没有正式的命名，所以我干脆借用“横向发展”的说法好了。

什么是横向发展呢？举例子来说，我们写个程序，深度发展关注的是让它速度更快、资源消耗更少，广度发展关注的是让它更合适与其它模块交互，甚至用更合适的语言编写这个程序。横向发展，则是让这个程序成为真正能用的程序，而不是实验室里的玩具。换句话说，“横向发展”是让程序更加“工业化”而不是“技术化”的发展。

我刚开始工作的时候，有一天提前完成了任务，喜滋滋地去向项目经理汇报。不料他看了代码之后，却把我劈头盖脸说了一顿：你以为你还是学生呢，给老师写个程序算出正确结果就完？你看你处理网络连接的部分，对服务器返回的异常信息，包括网络传输的各种意外都没有处理，谁向你保证服务器总是返回正确信息的？谁告诉你网络传输不会意外的？万一网络断了，你的程序就一直死循环吗？……

我必须承认他说的有道理，但也一时无可奈何。虽然在学校的时候写过不少程序，但老师都只看大致结构和结果，从没有问过“网络断线了怎么办”，也没有哪本教材专门讲过这方面的知识，所以自己一直也没想过。但是没想过归没想过，项目经理说的毕竟有道理，确实只有学生才会写出在理想环境下运行的程序。于是我开始有意识地学习和思考各种异常情况的处理，觉得讲究挺多，思路也因

此拓宽了不少。不久，还因为这方面的工作得到了项目经理的表扬，也深刻感觉到“横向发展”确实解放了自己。

后来换了份工作，我本来以为自己之前的经验可以被人赏识，却发现自己完全想错了。新工作对程序的要求更高、应用场景更严苛，只思考在程序内部怎么处理异常是不够的，还需要确保程序的持续运行，其运行状态持续可以记录、监控、分析，出现问题必须能在第一时间判断症结（而不是启动 IDE 去 debug）……为了做到这一切，既需要专门开发程序去监控自己的程序，又需要让原有程序能够被方便的监控，还不能泄露不必要的信息，所以在设计时又有更高的要求——当然，这些知识仍然是书上没有的。我写到最后才发现，虽然核心的功能并没有变复杂，但为了保证核心功能的稳定运行，程序本身的复杂度却上升了很多。这种要求，颇有几分类似小朋友的“横向发展”——但是小胖墩的重心终归要稳一些嘛，所以我把对程序员的这种要求称为“程序员的横向发展”。

或许是从工作开始就有机会重视“横向发展”的缘故，所以我长期以来并不认为这是严重的问题。后来的见识却刷新了我的认识：曾经有朋友告诉我，国内互联网行业某新兴领域排名三甲的公司，竟然连自己的服务器上跑的哪个版本的程序都不知道，开始我还当是笑话，后来才知道事实当真如此。小朋友的“横向发展”不讨人喜欢，许多程序员也忽视甚至讨厌“横向发展”，觉得这是在给自己找麻烦，他们认为，把核心功能写完，代码提交，往服务器上一扔，自己的工作到此为止了。至于其它方面，那就是系统管理员要处理的了。

如果你认真回忆，一定见过许多这样的程序：完全不处理意外情况，各种异常一股脑交给操作系统去处理，我甚至见过默不作声把所有异常都吃掉，假装没事继续运行的系统。也见过很多这样的程序：自动发送邮件的程序，不知道自己每天发了多少封邮件，消耗了多少流量，等到用户收不到邮件才知道出了问题；备份数据库的程序，不会记录每次备份的开始时间、结束时间、备份文件大小，直到硬盘满了才发现已经很久不能正常备份了；抓取数据的程序，不知道抓取的成功率、速度、消耗的流量，非要业务部门说数据很久没更新了才知道抓取失效了…… 其实这些功能通常都不复杂，但完成它们的程序，不管什么平台，什么语言，就是做不到稳定。每次出了问题都不能预先知道，又因为没有详细的记录，又要消耗无数的人力物力去解决。在一些稍微复杂的系统里，不少程序员每天的

工作内容就是这样的重复劳动，随之而来的是无休无止的抱怨，说工作毫无意义，没有机会学新东西…… 更糟糕的是，不少这样的程序员业余时间还在积极学习，希望在把语言工具掌握得更熟练，学会更多的语言和工具，却不知道问题的症结在于自己缺乏“横向发展”的意识。

我仔细回忆自己小时候，家长和老师会在一种情况下提倡“横向发展”，那就是要求身板像“豆芽菜”一样的同学多锻炼，成长结实一点。同样的道理，如果程序员觉得自己写出的程序像“豆芽菜”一样没有底气、不能放心，与其继续钻研新语言、新技术，倒不如抽出精力去“横向发展”一把。

原文链接：

http://www.luanxiang.org/blog/archives/1656.html?utm_source=tuicool